Titre: Title:	Software Tracing Comparison Using Data Mining Techniques
Auteur: Author:	Isnaldo Francisco De Melo
Date:	2017
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	De Melo, I. F. (2017). Software Tracing Comparison Using Data Mining Techniques [Mémoire de maîtrise, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/2733/

UNIVERSITÉ DE MONTRÉAL

SOFTWARE TRACING COMPARISON USING DATA MINING TECHNIQUES

ISNALDO FRANCISCO DE MELO JUNIOR
DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE INFORMATIQUE)
AOÛT 2017

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé:

SOFTWARE TRACING COMPARISON USING DATA MINING TECHNIQUES

présenté par: DE MELO JUNIOR Isnaldo Francisco
en vue de l'obtention du diplôme de: Maîtrise ès sciences appliquées
a été dûment accepté par le jury d'examen constitué de:

M. GAGNON Michel, Ph. D., président
M. DAGENAIS Michel, Ph. D., membre et directeur de recherche
M. ALOISE Daniel, Doctorat, membre

DEDICATION

יהוה

Χριστός

A minha mãe.

to God.
to Jesus.
to my Mother.

"Maria, Maria
É o som, é a cor, é o suor
É a dose mais forte e lenta
De uma gente que ri
Quando deve chorar
E não vive, apenas aguenta"
Milton Nascimento

"Maria, Maria
It is the sound, the color, and the sweat
The slowest and strongest dosage
Of the ones that laugh
When they should cry
They do not live, endure instead

A chi bene crede, Dio provvede.
To the ones that really believe, God will provide.
Italian Proverb.

ACKNOWLEDGEMENTS

I would like to thank all those who have supported me during my graduate studies, especially Professor Michel Dagenais, who gave me his support and advice for this research.

I acknowledge all my committee members: Professor Michel Gagnon and Professor Daniel Aloise to evaluate my research contributions.

Also, I would like to thank my colleagues at the DORSAL laboratory, for their valuable comments on my development. They have helped making my journey possible with their precious comments and considerations.

Special thanks to Gabriel A., Leticia R, and uncle Nacib Asseff, for their tremendous support during my studies and help in the several challenges faced.

I also would like to praise the Ecole Polytechnique de Montreal and the department of Computer and Software Engineering for giving me the opportunity of studying and working here.

Last but not least, I would like to acknowledge my mother, Maria Pereira de Assis, for her imperative support throughout my journey. Infinitude.

We came, We saw, God conquered

RÉSUMÉ

La performance est devenue une question cruciale sur le développement, le test et la maintenance des logiciels. Pour répondre à cette préoccupation, les développeurs et les testeurs utilisent plusieurs outils pour améliorer les performances ou suivre les bogues liés à la performance.

L'utilisation de méthodologies comparatives telles que Flame Graphs fournit un moyen formel de vérifier les causes des régressions et des problèmes de performance. L'outil de comparaison fournit des informations pour l'analyse qui peuvent être utilisées pour les améliorer par un mécanisme de profilage profond, comparant habituellement une donnée normale avec un profil anormal.

D'autre part, le mécanisme de traçage est un mécanisme de tendance visant à enregistrer des événements dans le système et à réduire les frais généraux de son utilisation. Le registre de cette information peut être utilisé pour fournir aux développeurs des données pour l'analyse de performance. Cependant, la quantité de données fournies et les connaissances requises à comprendre peuvent constituer un défi pour les méthodes et les outils d'analyse actuels. La combinaison des deux méthodologies, un mécanisme comparatif de profilage et un système de traçabilité peu élevé peut permettre d'évaluer les causes des problèmes répondant également à des exigences de performance strictes en même temps. La prochaine étape consiste à utiliser ces données pour développer des méthodes d'analyse des causes profondes et d'identification des goulets d'étranglement.

L'objectif de ce recherche est d'automatiser le processus d'analyse des traces et d'identifier automatiquement les différences entre les groupes d'exécutions. La solution présentée souligne les différences dans les groupes présentant une cause possible de cette différence, l'utilisateur peut alors bénéficier de cette revendication pour améliorer les exécutions.

Nous présentons une série de techniques automatisées qui peuvent être utilisées pour trouver les causes profondes des variations de performance et nécessitant des interférences mineures ou non humaines. L'approche principale est capable d'indiquer la performance en utilisant une méthodologie de regroupement comparative sur les exécutions et a été appliquée sur des cas d'utilisation réelle. La solution proposée a été mise en œuvre sur un cadre d'analyse pour aider les développeurs à résoudre des problèmes similaires avec un outil différentiel de flamme.

À notre connaissance, il s'agit de la première tentative de corréler les mécanismes de regroupement automatique avec l'analyse des causes racines à l'aide des données de suivi.

Dans ce projet, la plupart des données utilisées pour les évaluations et les expériences ont

été effectuées dans le système d'exploitation Linux et ont été menées à l'aide de Linux Trace Toolkit Next Generation (LTTng) qui est un outil très flexible avec de faibles coûts généraux.

ABSTRACT

Performance has become a crucial matter in software development, testing and maintenance. To address this concern, developers and testers use several tools to improve the performance or track performance related bugs.

The use of comparative methodologies such as Flame Graphs provides a formal way to verify causes of regressions and performance issues. The comparison tool provides information for analysis that can be used to improve the study by a deep profiling mechanism, usually comparing normal with abnormal profiling data.

On the other hand, Tracing is a popular mechanism, targeting to record events in the system and to reduce the overhead associated with its utilization. The record of this information can be used to supply developers with data for performance analysis. However, the amount of data provided, and the required knowledge to understand it, may present a challenge for the current analysis methods and tools.

Combining both methodologies, a comparative mechanism for profiling and a low overhead trace system, can enable the easier evaluation of issues and underlying causes, also meeting stringent performance requirements at the same time. The next step is to use this data to develop methods for root cause analysis and bottleneck identification.

The objective of this research project is to automate the process of trace analysis and automatic identification of differences among groups of executions. The presented solution highlights differences in the groups, presenting a possible cause for any difference. The user can then benefit from this claim to improve the executions.

We present a series of automated techniques that can be used to find the root causes of performance variations, while requiring small or no human intervention. The main approach is capable to identify the performance difference cause using a comparative grouping methodology on the executions, and was applied to real use cases. The proposed solution was implemented on an analysis framework to help developers with similar problems, together with a differential flame graph tool.

To our knowledge, this is the first attempt to correlate automatic grouping mechanisms with root cause analysis using tracing data. In this project, most of the data used for evaluations and experiments were done with the Linux Operating System and were conducted using the Linux Trace Toolkit Next Generation (LTTng), which is a very flexible tool with low overhead.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ACRONYMS AND ABBREVIATIONS

IETF	Internet Engineering Task Force
OSI	Open Systems Interconnection
CG	Call Graph
ANOVA	Analysis of Variance
CCT	Calling Context Tree
CPU	Central Processing Unit
CTF	Common Trace Format
ECCT	Enhanced Calling Context Tree
ETW	Event Tracing for Windows
ELF	Executable and Linkable Format
LTTng	Linux Trace Toolkit Next Generation
SVM	Support Vector Machine
FSM	Finite State Machine
IDS	Intrusion Detection System
SDG	System Dependence Graph
PCFG	Probabilistic Context Free Grammar
DFG	Differential Flame Graph
SLOs	Service Level Objectives
DCA	Directed Acyclic Graphs
ANTLR	ANother Tool for Language Recognition
ETW	Event Tracing Windows
PADBI	Performance Anomaly Detection and Bottleneck Identification

ML	Machine Learning
OAP	Aspect-Oriented Programming
DTW	Dynamic-Time Warping algorithm
SCFG	Stochastic Context Free Grammar
EDT	Enhanced Dynamic Tree
CCRC	Calling Context Tree Ring Charts
PARCS	Performance-Aware Revision Control Support
BCEL	Byte Code Engineering Library
ICFG	Interprocedural Control Flow Graph
RGG	Red Green Gray Differential
SSE	Sum of Squares Error

CHAPTER 1 INTRODUCTION

Tracing, profiling and debugging are essential tools for software development and performance evaluation. Those tools currently face challenges related to many-core and cloud based systems, because of the enormous amount of data to be analyzed. It is expected that with the rapidly increasing scale and complexity of these systems, we will soon exceed the limits of human capability in terms of quantity of data or time to evaluate it.

In this context, there is an increasing demand for automation and reduction of analysis time. This tendency creates an interesting opportunity for developing automated analyses and methods. Accordingly, this graduate research explores several automated mechanisms for performance analysis through tracing and profiling techniques.

This document presents the state of the art in terms of analysis tools related to tracing and profiling, by improving current tools and mechanisms. It proposes data mining techniques to explore and solve problems without the requirement of very specific knowledge of underlying processes of the system, thus reducing the analysis time of the data.

1.1 Challenges in tracing analysis

The first challenge in tracing analysis is to mine meaningful information from the tracing data. A possible way to solve this issue is to group or cluster the tracing data or to add a level of abstraction, to explore and measure its behaviour. Using clustering, the challenge becomes the classification of the data without supervision, and how to efficiently apply this technique to measure the causes of the performance issues.

The second main challenge is identifying with certainty the root cause for a specific performance issue. This problem is related to the fact that, considering clusters that delimit different executions, even with clear properties differentiating them, these differentiating properties are not necessarily the underlying cause for the performance difference. This challenge can be summarized as: association does not mean causation. Thus, even if a performance metric (such as cache misses or page-faults) is capable of explaining all the differences between the groups, it is not necessarily the cause of this difference.

In summary, these challenges may constitute a real barrier for performance evaluation and diagnosis. They have been partially solved with the current solutions and, despite the success of tracing techniques, there are still opportunities for further enhancements in terms of data analysis.

1.2 Elements that impact performance

Several elements compose the performance of an application, especially in the current programming languages like C and C++. The use of prefetching mechanisms and compiler optimization might considerably impact the software performance. Out-of-order executions can impact the performance as well.

The use of parallelism architectures significantly improved the overall performance of computers, and consequently software applications. For example, the Skylake and Kaby Lake architectures bring hyper-threading, large L3 caches, and four or more physical cores. However, this performance improvement is not linear, since it will follow the Amdahl's law of parallelism.

The limitation in the number of cores and their communication capabilities, as well as other hardware limitations that processor manufacturers are facing, inexorably is calling for improvements in software algorithms and related features, as explained by [125]. Better algorithms and data structures will be required especially in large industrial applications. Tools such as profilers and debuggers can be used to improve small parts of a software application or find bugs, respectively.

At the same time, an application is not executed in isolation on a system. The operating system may have a strong impact on the execution efficiency and cannot be neglected. Consequently, performance tools such as profilers have a limited analysis capability in the overall context of an application, and the use of tracing is normally required to account for interactions with the operating system. Therefore, as a consequence of the software requirements, the context of utilization of an application and the amount of resources (and features) that might impact the software performance, detailed and efficient performance analysis methods will be required.

In summary, this work combines the current profiling and tracing tools to target these specific elements of performance evaluation.

1.3 Objectives of the research

The main objective of this work is to reduce the need for human intervention in the analysis of the system performance. The methodology to fulfill this objective is a comparative methodology to segregate the data in clusters and later to compare the fast and the slow executions.

The collection of the data is done using tracing and profiling techniques, combined with data mining, heuristics and statistics tools. The developed methods were able to find real associ-

ations in data with metrics and consequently indicate root causes of performance issues.
This work focuses on the Linux operating system and its tools, such as LTTng and Perf counters. Nonetheless, the heuristic approach could be applied to any system that is able to describe the behaviour of a system using metrics.
More specifically, the objectives can be summarized as follows:

1. Reducing or avoiding the requirement of human intervention in tracing data analysis
2. Improving the current methods to find root causes using metrics
3. Developing an optimized solution for the proposed approach

1.4 Specific Problem

As specific problem, we want to compare several executions and uncover the main reason for any anomaly, i.e. find the specific root cause of software anomaly using different metrics such as performance counters. Thus, several runs of a program are executed while the behavior of each run is recorded. If those executions present substantial differences, the comparison can generate information about the underlying location or cause, and provide hints for later improvement.

1.5 Assumptions

There are two main assumptions in this project:

1. There are groups of similar executions, and thus the groupings can help uncover the causes for differences
2. Only the properties within the groups explain the differences

The first assumption is taken from the work of [28], the methods studied in this research aimed to find groups, which can be associated to software behaviors. The second assumption, the properties, imply that performance metrics can explain the anomalies in the system. Otherwise, it is anyhow not really possible to take in consideration hidden factors that influence the behavior of the system.

1.6 Proposed Solution

Aiming the objectives stated above, this research explored several data mining and statistical methods to mine information collected with profiling approaches. Focusing on the low-overhead, we used a tracing approach for the user-space applications. Later, the data is abstracted in a data structures, ECCT or EDT, which combine profiling data with performance metrics in its nodes, using a sampling technique.
The next step is to apply the data mining approaches to mine data, basically to segregate the similar nodes of the tree and compare them. A comparative approach to the groups can be seen as similar to using statistical means or the Apriori approach.
The comparison of the groups will give the better and worst groups and the indication of the cause of the differences between the groups. This indication can be used as a hint to the need for improving some inner algorithm within the application, or to change a system parameter for instance a priority level in case of a scheduling interference problem.
The solution can be extended to consider trace correlation and other trace abstractions. Also, the use of other data mining techniques can reduce the amount of time requested for the analysis, although the assumptions will continue to be the same.
We follow a similar pattern as TraceCompare, which requires manually selecting one group for each metric, by applying automatic grouping techniques for each metric separately. Consequently, we applied a bi-dimensional analysis (e.g. function id versus a specific metric) instead of multi-dimensional, which would be the next step and would fully benefit from data mining methods.

1.7 Solution result

This research, i.e. the root analysis investigation, aims to find metrics, such as instruction count, page-faults, cache misses, that indicate the possible cause of a performance issue. The analysis tool will not indicate an hypothesis, in fact it will produce an estimation that the metric is the cause of the performance difference. The next step is left to the user, which can make a claim stating that the metric is or is not partially responsible for the performance issue. The result of the proposed solution will not be an hypothesis, as defined by [11]. Rather, it will be an evaluation, which considered only the metrics already there. Those metrics come from the tracing and profiling tools and rely on those tools to provide accurate data about the application. The clustering or grouping process does not produce more information that what is available.
In summary the solution may or not produce a sound truthful evaluation, depending on two

main statements:
(i) The profiling/tracing is able to measure the system with precision (e.g. minimal overhead)
(ii) The system behavior can be represented with the selected metrics (e.g. a task does not use resources that cannot be measured with the metrics)

1.8 Use of the solution

The dissertation brings a compilation of the research focusing on several aspects of data mining, considering many tools such as machine learning and statistical evaluation. The application of the methods requires static or dynamic instrumentation, as well as performance data.
The proposed solution, a grouping mechanism to track root cause analysis, can be used in any context where the runs can be separated in executions, with associated performance counters measurements. Scenarios like this can be found in C/C++ userspace applications, but also in web application, where the server can be instrumented to record requests and their performance counters.
The mining algorithms can then be applied to cluster the anomalies, or isolate abnormal executions, and compare them using profiling techniques. The result of this classification can be used in fuzzy groups, i.e. not just two totally fast and totally slow performance groups, and this can bring more information for the analysis in general.
The proposed RGG Differential Flame Graph can reduce the ambiguity for equal performance functions and is suitable for fast and slow comparison. Finally, an heuristic evaluation can be used, combined with other clustering or grouping tools such as k-means, to reduce the need for human intervention. Specifically, in terms of k-means algorithm, its use gives the optimal number of k for the clustering algorithm to work.
The solution can be applied directly in real applications. This tool may be used in regressions tests, where unit tests, profiling tools and debuggers can not reveal the causes for performance deviations. Further details will be provided in the the article. To help in the understanding of the solution, we will present some definitions intrinsically related to the research, in the following section.

1.9 Basic Definitions

1.9.1 Anomaly

Anomalies can be simplified as discrepancies in the expected distribution, or as a point or group of data points lying outside an expected normal region, as outlined in [23].

However, this research focuses on a specific kind of anomalies, collective anomalies, but may also consider point anomalies. According to the definition of anomaly in [16] it is related to a group of discrepancies in terms of standard behaviour. The collective discrepancies may indicate a series of conditions related to the observed software.

1.9.2 Execution

Execution is any run of a program that can be traced, and the performance data that can be used to measure its behaviour. Executions can be grouped so that they can be compared systematically using several techniques. Inside an execution, several performance parameters can be present. In this research, we focus on comparison methods for groups of executions, which can be compared using several statistical techniques. An execution can be formally defined as a collection of finite metrics in the Natural numbers (M1, M2, .., Mn).

1.9.3 Profile

Profiling is the dynamic analysis of a program. Broadly, there are two main types of profilers: the ones that count the number of invocations and the ones that display the a time measurement about the statements and routines. An example of profiler is gprof, which counts the running time of routines [56].
The profiling mechanisms can be used to analyze in depth the core of an application or the system, specifically the stack frames of the application.

1.9.4 Debug

Software debugging is the general process of finding and resolving software issues. There are several strategies for debugging such as isolating the problem and verifying the assumptions. Many tools can be used to debug. A well known debugger is GDB, the Gnu Debugger, which can debug several languages for example Ada, C, C++, Objective-C and Pascal.

1.9.5 Instrumentation

Instrumentation can be defined as a technique for inserting trace statements at some locations within the code. Instrumentation can be done dynamically and statically. The first, dynamic instrumentation, offers the possibility of changing the code instrumentation during execution time. Static instrumentation is the process of inserting instrumentation directly in the source

code [56]. In LTTng, probes are inserted in the kernel code, then the kernel is compiled, and the system is rebooted with an instrumented version [132].
Instrumentation can be simplified as the process of inserting extra code in the application to measure its behaviour. Instrumentation can be performed at various stages: in the source code, at compile time, post link time, or at run time.
There are two main possibilities for dynamic instrumentation of code: probe-based and jit-based. This first one, probe-based, is used by Dyninst in [99], Vulcan from [35], and [33]. The second instrumentation approach is jit-based, such as in Valgrind [95], Strata [113], DynamoRIO [130], Diota [82], and Pin in [80].

1.9.6 Tracing

The concept of tracing can be defined as a very fast system-wide fined grained logging mechanism. Unlike logging, which deals with high level records about the system, tracing records the low-level events of the system. This record can then be used for other analysis mechanism, such as sequence matching, trace abstraction and visualization tools.
The understanding of a complex multicore system it not trivial. By browsing a list of events, for example, the trace files are analyzed using a program that processes the traces. Those tools will generate graphical or textual reports, for example about several resources of the system, including: process status, amount of data read or written, latencies and Interruptions.

1.9.7 Events

Events here are defined as a record containing: a timestamp, a type and some arbitrary payload. They can also be used to highlight the entries and exits of functions in a software application and be combined with performance metrics. They can be used to measure the current state of an application [52].

1.9.8 Metrics

The concept of metric here is the hardware or software performance runtime information and can be used to gather several aspects of the system behavior [52]. The term metric used in this work is mostly related to hardware and software metrics, typically collected using Perf. The information monitored is a current metric of the system, and it can be recorded in the nodes of a data structure (i.e. in the nodes of the ECCT or EDT) using sampling techniques.

1.9.9 Root Cause Analysis

The term Root Cause Analysis in this research is restricted to the analysis of causes for delays in executions, i.e. specific reasons for delays in executions. This identification of performance degradation causes might use several methods, including statistical analysis, [20].

1.9.10 Cluster Analysis

Cluster analysis can be defined as the process of subdividing the data in similar subsets, requiring a distance (similarity) metric computation method. In [72], a statistical analysis is performed for server analysis. In this research we aim to create clusters, which are later used to compare the performance of the executions in terms of metrics. The groups can be described as sorted or unsorted collections of executions.

1.10 Outline of the research

In Chapter 2, the Literature Review is presented, which describes the main aspects of this research. Detection and diagnosis tools are surveyed, with details. Moreover, different methods for data mining are discussed, with a compilation of their pros and cons and where they are typically used.
The next chapter, Chapter 3, Methodology, presents the hypothesis, assumptions and details about the methods applied.
Later, Chapter 4 contains the Article «Performance Analysis Using Automatic Grouping». This article presents an automated solution, which includes data collection using tracing mechanisms, and data mining methods to compare groups of executions. The proposed approach is used for real software applications in C/C++. The article was submitted to the Journal Software: Practice and Experience.
Chapter 5 is the General Discussion and Complementary Results, discussing details of the results and methods. A comparative approach is described. Finally, Chapter 6 is the Conclusion, which includes a summary of the work, suggestions about the method and its specificities. It also discusses limitations and possible future work.

CHAPTER 2 LITERATURE REVIEW

Performance is an important and challenging aspect of the development of software and web applications. Some of the challenging aspects lie in distributed and many core systems, which increase the computational power and the complexity of those systems. In this matter, complementary tools such as profilers and tracers, need to be used in order to minimize their influence in real systems when measuring their behavior. Those tools and methodologies are used for performance analysis with different aims.

This chapter provides insight on tools and methods in terms of collecting and analyzing performance data. It also describes techniques for comparing traces, and manual and automated analysis of tracing. Finally, the required knowledge to understand in depth our proposed solution is presented in this chapter.

Following this, useful techniques and tools for tracing will be presented in Sections 2.1 through 2.6. Then, analysis methods are presented, classified as automated and manual. Later, the current challenges in tracing analysis, and the associated dynamic data structures, will be presented. Finally, the proposed solution is introduced.

Overall this part aims to present the tools with critical judgment about the current data mining techniques and how they apply to this research.

2.1 Static analysis and dynamic analysis

There are two generic and complementary kind of tools to investigate software issues: static analysis and dynamic analysis, as discussed later in more details. Each of these tools have strengths and weaknesses.

The first kind of analysis, static analysis, presents tools that use source code and documentation for their evaluation. Those methods include source code analysis tools [104] and other tools such as [63] and [71].

There are several ways to statically analyse code, according to [5]. The two main characteristics of static evaluation are the nature and the depth of the analysis. Source code analysis tools include several tools such as Semantic Diff, Better Code Hub and FindBugs.

The second tool Better Code Hub, [63], does a complete analysis of several aspects of the source code of applications, focusing on quality measurements and code metrics. The tool also uses definitions of guidelines for code maintenance, such as writing small units of code and doing automated tests. Although not scanning the code itself, but the bytecode, another tool is FindBugs, [118], which does a search for bug patterns. It requires compiled class files

to find the bugs from patterns, and it creates a scale for the possible bugs occurences, from one to four, where one is the worst. The reported false positive rate is about 50 percent [123]. This tool also has several configurable settings. FindBugs does not find all bugs, since context sensitive bugs and security bugs, which involve the dynamic behavior of software, are not tracked.
Some tools use pattern languages to describe bugs, as PMD [120]. PMD uses a set of predefined rules in classes. Examples are Double Checked Locking pattern and Unconditional If statements.
Static analysis, however, is limited to the code and consequently is not able to reproduce exactly the behaviour of complex and multi-threaded systems. The alternative is dynamic analysis, that basically operates by executing a program and observing (and recording) the execution. The analysis of this runtime data can be used for performance evaluation, for example, but also for software testing. Examples of dynamic analysis tools are debuggers, profiler and tracers.

2.2 Profilers

The first kind of tools, Profilers, allows to verify where the program spent its time and which function called which other function (call hierarchy) while it was executing. This information can be used to identify the faster and slower parts of the code and improve the performance of the application. An example of profiler is a call graph profiler that shows the call durations. Gprof is a profiler from GNU Compiler toolchain. Gprof was originally proposed to help the user to evaluate alternative implementations of abstractions.
Gprof presents a call graph of the application, which represents the caller-callee relationship, also showing the number of times that each function was called. The implementation in gprof issues a call to the mcount function upon each function entry. This function is responsible for creating a table using the stack frame of the application. Another tool called OProfile, is a set of monitoring tools related to profiling. It was implemented using a script called opcontrol and a daemon called oprofiled, and is able to collect information for all parts of the system: kernel, shared libraries and binaries [119]. This tool uses a mechanism called differential profiles that is able to compare profiles by percentage and has a declared overhead impact of between 1% to 8%.
However, profilers add a non-negligible amount of overhead to the system, because of the requirements of the code that need to be instrumented. The process to analyse the data is separate from running the program itself, and is performed afterwards. The distribution

of the function durations helps in identifying the contribution of each function to the total execution time, but does not necessarily reveal infrequent performance problems, such as occasional latency constraints violations.

2.3 Debuggers

The second kind of tools, debuggers, enables source code analysis during the execution and aid in understand program execution. This is achieved by dynamically adding breakpoints or tracepoints points at arbitrary locations in the code, such as when the program crashes. One example is GDB, the GNU Debugger. It supports several languages, such as C and C++. GDB starts as a separate server process and executes the user commands. There are three kinds of breakpoints in GDB: breakpoints, watchpoints and catchpoints, all of them involving stopping the normal execution of the program [121]. The process of debugging is to watch the behavior of those breakpoints, analyzing concomitantly the source code of the application.

Debuggers, however, increases considerably the overhead of an application, because it needs to stop the execution of the application when a certain condition occurs. This kind of analysis requires the addition of breakpoints in the code. Also, since they present a snapshot of the application execution, they often cannot find complex component interactions issues [122]. Finally, GDB is primarily designed for fixing bugs rather than tuning performance.

2.4 Tracing

Among the current dynamic analysis tools, tracing is a technique that incurs lower overhead, and consequently can be used to measure with accuracy several properties of the application, such as its performance. Unlike logging, which deals with high level records of the system, tracing records the low-level events of the system. This record can then be used by other analysis mechanism [25].

Tracing can be classified based on two main aspects: its functional aspect and its target domain. The first aspect, the functional, states that tracing instrumentation can be inserted statically or dynamically. The first, static tracing, requires source code modification and recompilation of the target binary/kernel. This is unlike dynamic tracing, which inserts the tracepoints directly in the running process or binary.

Tracing can also be divided into two target domains: userspace applications and kernel

applications. The first domain, userspace, concerns applications executed in userspace, i.e. the portion of memory in which a user executes its processes. In this domain, the system calls can be recorded.
The later domain, kernel space, concerns the system core, the kernel. In this domain, several tools, such as LTTng, already benefit from predefined tracepoints in the kernel code.

2.5 Tracepoints

Tracing involves the addition of tracepoints in the program to measure its behaviour. Each tracepoint is associated with an event and can be used to record the behavior of the system, without the need to create new tracepoints in the source code.
In this way, there are two types of tracepoints: static tracepoints and dynamic tracepoints, [26].
The first type, dynamic tracepoints, do not have any impact when not activated. Linux has had dynamic trace functionality for a long time in the form of probes, kprobes, jprobes, and kretprobes. By using probes, it is possible to dynamically enable new tracepoints in the kernel, and consequently collect debugging and performance information non-disruptively [136]. Kprobes, specifically, provide a kernel API for placing probes at kernel instructions and they can be exploited directly via a kernel module, or via systemtap which provides a high level scripting language. Kprobes basically is a set of handlers to be placed at specific instruction addresses [55], where a post-handler and a pre-handler is defined. The handlers can be used for multiple probes. The process is the defined as follows: when the instruction is executed, the pre-handlers are executed before the execution, while the post-handlers are executed right after the execution of the instruction. Also, it is possible to use GNU debugger (GDB) tracepoints to insert dynamic tracepoints in the system [116].
The second type, static tracepoints, use the already present static tracepoints in the Linux mainline code for collecting data. There are several ways to use the static tracepoints, via commands or in-kernel modules. Usually, static tracepoints are faster than dynamic tracepoints [52].

2.6 Tracing tools

In kernel space, as described above, the main available tools are the following: LTTng, Perf, eBPF and Ftrace.
The first, LTTng, the Linux Trace Toolkit Next Generation, was proposed by Desnoyers and

Dagenais as a tracer to extract information from the Linux kernel, user space libraries and from programs by running a recompiled instrumented version of the kernel. It was created with the objective of minimizing the impact of the instrumentation in the kernel,[27].
The LTTng main components are the daemons, the session and the ring buffer. The session daemon is responsible for interfacing with the sessions and the overall component control, while the second, the consumer daemon, writes the data to CTF traces. The session is the communication mechanism between the user and the session daemon, enabling multiple records at the same time. Finally, LTTng uses configurable circular buffers, which have configurable features for recording the kernel events and can be tuned according to the user needs. The ring buffers are associated with a channel.
LTTng provides three main modes for trace recording: normal, snapshot and live mode [36]. The first mode is the normal mode, i.e. reading and writing in the ring buffer, by the session and consumer daemons. The second mode is the record of a snapshot of the trace, periodically saved in the buffer and sent to the (possibly remote) reading system through the relay daemon. The third, the live recorder mode, is where a system viewer can be used to analyze the information as it is generated. Also, LTTng is currently able to trace Java and Python applications using similar techniques. The tracing data generated by LTTng is output as Common Trace Format (CTF). This format can then be read using Babeltrace or parsed via scripting languages, such as python bindings, [36].
The following tool, Perf tools, or perf, is a mainline Linux utility that was developed to measure several software and hardware metrics on the CPU. It is a profiler tool for Linux 2.6+ based systems that abstracts away CPU hardware differences in Linux performance measurements and presents a simple command-line interface. Perf is based on the perf events interface exported by recent versions of the Linux kernel. This research uses the perf tool for the construction of the dynamic data structures to measure the system behavior.
The functionality of Perf was subsequently extended to interface with the macro TRACE_EVENT() and therefore access the Linux kernel trace points. It is possible to use it to generate statistics on the number of times a tracepoint is executed, for example, or to analyze the number of events that a processor will have recorded during a time period. It can be used to record profiles on per-thread, per-process and per-cpu basis using a sampling approach.
A new tool that evolved recently is eBPF, the enhanced Berkeley Packet Filter [75]. It is being integrated in the Linux mainline and allows several filtering packages but also custom analysis filters to be achieved efficiently, [59]. EBPF can be used for computing, with low-overhead, latency histograms and heatmaps that can be used in detailed performance analysis and through visualization tools. This tool is extremely versatile and can be used to create programs, eBPF programs, which can be attached to kprobes with low-overhead and

populate the eBPF maps, as explained in [60].

Finally, Ftrace is well known mainline tool for Linux tracing. It was designed to find the specificities of the operations inside the kernel. It can be used for debugging or analysing latencies and performance issues that take place in kernel space. Ftrace uses the TRACE_EVENT() macro, [69].

Although Ftrace is typically considered the function tracer, it is indeed a framework of several tracing utilities on the kernel side. For example, it can be used for latency tracing, to examine what occurs between interrupts disabling and enabling, for analyzing preemptions and for other analyses. This tracer is used to dynamically trace functions in the kernel, to help debugging kernel detailed operation.

Another tracer, SystemTap, allows tracing in kernel space, as well as in user space, through the use of dynamic instrumentation, i.e. uprobes and Kprobes. SystemTap relies on a scripting language, that provides flexibility to the user, and comes with an interactive GUI. To run those scripts, once written, they are compiled into a module and loaded into the kernel. Although this tool offers a great flexibility, it is not suitable for the collection of a large number of events, because it carries a larger overhead than LTTng.

From the userspace side, the following tools can be used:

The first tool, LTTng-UST, is the counterpart of LTTng (kernel) but for user-space. For this purpose, the tool can be used statically in the code, by inserting tracepoints or by the use of instrumentation mechanisms during the compilation phase. Compiler inserted instrumentation, and manual tracepoint insertion, both have drawbacks. Manual tracepoint insertion in the source code demands time and might not be done in all the functions. Compiler inserted tracepoints are limited to specific generic locations, such as functions entries and exits, and the indiscriminate insertion in every function might cause an excessively high overhead. LTTng-UST provides a simple way to introduce tracepoints through the compiler. The GNU Compiler flag *finstrument-functions* add a call to specific trace functions at each function entry and exit, similar to a profiling mechanism, and the trace functions are added to the binary using the LD_PRELOAD dynamic linking mechanism.

In terms of runtime efficiency, LTTng-UST has a mechanism to avoid returning to the kernel, separating the traced process from the consumer daemon. The traced application is responsible for writing the events into the shared memory ring buffers, which are subsequently read by the consumer daemon. The consumer daemon is specific for user-space tracing since it is a different process than for kernel tracing. The application directly writes the events in memory using lockless operations and consequently does not pass through the kernel of the operating system.

Once a trace buffer is filled, it uses a non-blocking control channel to wake the consumer

daemon, and then change to a another buffer to continue recording the events. Then, the consumer daemon will save to disk (or to the network) the shared memory buffer content, releasing it for reuse. This reduces the communication between the consumer daemon and the traced application.

Another tool for user-space, Perf, can be used directly with user-space code, where code to read the performance counters is added directly in the code through the use of a library in the source code. In this approach, the performance counters can be recorded using a sampling approach, where the increment for a specific metric from the beginning to the end of a code section execution can be computed and recorded. This tools is extremely flexible and is able to measure many system metrics, both software and hardware, including instructions executed, page-faults, cache-misses and so on.

Another user-space tool, uftrace, developed by [77] can also be used to trace and analyze execution of programs written in C/C++. It was heavily inspired by the ftrace framework of the Linux kernel (especially function graph tracer) and supports user-space programs. It supports various kinds of commands and filters to help in the analysis of program executions and performance analysis.

Uftrace is also able to trace kernel functions, requiring root privileges and by enabling the CONFIGFUNCTION_GRAPH_TRACER=y option in the kernel.

Finally, SystemTap can be used as well for user-space tracing through DynInst. For tracing in user-space it uses a DynInst mutator [99], which can be activated through *stap*, the SystemTap command line tool. The SystemTap GUI can be used to edit the scripts. It is also possible to access tracepoints defined for DTrace. The traces produced by SystemTap are in plain text.

In terms of performance, however, it can be slower than solutions which use static tracing, for example LTTng, especially in multi-threaded environments. Apart from kernel and user-space Linux tools, there are other tracing tools highlighted below:

The Google Chromium browser, which is highly popular throughout the world, has a tracing feature. The browser provides a method to trace its application and analyse its performance, using an embedded tracing system, [54]. This is a complete solution at the application level, containing tracing and analysis tools, which is platform-independent.

The tracing mechanism is split in two: trace recording and trace consumers. Telemetry is a tool that can be used to trace and analyse this application, i.e. produce the traces while the Chrome DevTools application is a way to consume it [19].

The advantage of this technique is portability, since it works independently of the operating system or version. However, the portability of this system, due to its Java Script interface, is also the source of some limitations, notably a size limitation for the tracing data. Indeed,

the Chromium traces have a limited size, covering up to a few seconds and containing only a small number of events. Moreover, the sampling rate in Chromium may also cause issues for the analysis of its data, as described in other work [1].
Although related to cloud tracing, end-to-end tracing tools have been developed for quite some time. This technique basically enables the tracing of a request from the client-side to the backend, as defined in [126]. End-to-end tracing tools have been widely adopted by several companies as summarized in [81]. Among those end-to-end tracers, a useful tool is X-trace, [43], which is a diagnosis tool based on tracing and works through metadata insertion, aiming to target all nodes within a networked application. The main purpose of this tool is to reconstruct the task tree of a distributed task, encompassing all sub-operations making up the initial task, that is the set of network operations associated with the observed task. This is achieved by adding metadata in the application. This information is disseminated through the nodes of the application to measure it. X-trace covers several scenarios of application, such as web servers tracing. This tool is important to highlight, considering the several diagnosis mechanisms that require end-to-end instrumentation, such as Spectroscope.

2.7 Challenges in tracing analysis

The use of tracing for performance analysis can be extremely efficient, in terms of collecting and analysing data. However, there are some challenges regarding three main aspects: the size of the trace, the complexity of the data and the data analysis. Those aspects are discussed below: The first challenge is related to the the amount of data generated by tracing. This comes from the fact that tracing can record events at low-level and, consequently, many events are generated at kernel level, [52]. This amount of data cannot be easily used, compared, or analysed, even with pattern finding or comparison mechanisms. Some aspects of pattern matching can be improved upon, as emphasized by [92].
The second challenge related to the analysis of tracing data is the inherent complexity of the data [52]. Since the data covers many aspects of the system, the number of events may require that several techniques be applied, to mine the information from the enormous amount of data. The frequency of occurrences of some events in a system also plays a role in this challenge. For instance, an important event may happen only once in a thousand times or more, and consequently some techniques such as pattern mining might not be adequate in such situations.
The last challenge resides in the difficulty to analyze the cases, since a deep understanding of the underlying mechanisms of the system and applications are necessary. For some perfor-

mance issues, the root cause of the problem is unknown, and profiling all the application may not be trivial. Tracing can be used, but the main point in this issue is to find the meaningful associations between a singular and specific cause in the trace data with the source code modification that triggered it. For this challenge, a high level data structure, such as a call graph or calling context, combined with performance data, might be used [31].

2.8 Performance Metrics

Hardware and Software performance metrics can indicate key information to describe the state of a system [57]. Two examples of metrics are latency and throughput. The first can be used to describe the response time of an application, that is the time for any operation to complete such as a database request. The second metric, throughput, is the rate of work performed, for example the number of web requests served per unit of time.
System resources can be measured through hardware and software metrics, and the performance of the application that uses those resources can be measured indirectly. System resources might include physical components such as the CPU, memory, disk, caches and network, but also virtual components such network connections, sockets, locks, file handles or descriptors [57].
From this detailed information about the system, such as process scheduling and memory management, it is possible to abstract and control several behavioral aspects of an application, as well as of the system. Several utilities like top, ps and htop, can display such information live, from the system, and may be used by system administrators to measure the system behavior.
In terms of tracing techniques, performance metrics can be recorded using a special recording mechanism that writes the performance metrics using perf_events directly in the system. In this case, the metrics must be defined a priori, to be enabled in the tracing environment. Besides recording by this approach, it is also possible to use the system dump to record the metrics of the system as they are recorded.
Through the utilization of the approaches described above, it is possible to solve performance issues by correlating events with the performance metrics, or even with the system state.

2.9 Performance Anomalies

There are several classifications for software anomalies. One distinguishes three kinds [16], others consider four kinds of anomalies [67]. The anomalies can be punctual anomalies, collective anomalies and contextual anomalies.

The first kind of anomalies, punctual anomalies, represent an outline point in the expected values of an application. This definition is broad and might consider a statistical metric such as the distance from the average. For example, one standard deviation or more from the average may be considered an anomaly. Those kinds of anomalies might be the spikes in one application that might not affect more than one execution, but the impact on this single application may be considerable.
The second kind, collective anomalies, are anomalies that occur not just once but in groups, i.e. a group of outliers that, as a group, impact the application. According to the definition of [67], a special group of those anomalies occurs at a regular time interval, and consequently follow a pattern.
Finally the last kind of anomalies are those that occur because of the context of the application, called contextual anomalies. Those anomalies can be system problems, for example IO-bound or CPU-bound situations. Contextual anomalies are related intrinsically with the infrastructure of the application.

2.10 Trace analysis

There are different techniques to analyse trace data, the two possibilities are detailed below.

2.10.1 Manual Analysis methods

There are mainly three ways to analyze trace data, as explained below: The first, using sequence detection, which includes pattern matching techniques, was used in several ways in different contexts. Most of the aforementioned tools use this technique to detect repeated contiguous sequences of trace events, and to generate abstract and compound events [18]. [78], and [86] use pattern matching techniques to generate abstract events from the LTTng kernel trace events. Pattern matching can also be used in intrusion detection systems [65]. Some language features are used for matching in OCaml as described in [131]. For example, STATL models in [34] use signatures in the form of state machines, while in [22], signatures are expressed as colored petri nets (CPNs), and directed acyclic graphs (DCA) are used to extract security specifications. [7] presents an approach for malware detection using the abstraction of program traces. They detect malwares by comparing abstract events to reference malicious behaviors. Although not related to tracing, in [45], a behavioral diagnosis technique is used, based on an algorithmic approximation matching approach, which is a NP-complete problem; it is somewhat similar to what was achieved by Matni. In addition, a similar approach was explored by [44], analysing blocked processes on multi-core systems. This work was implemented in the former LTTV tool as the Delay Analyzer.

However, most of the techniques explained above use pattern-matching and have defined their patterns over trace events. They did not consider using the modeled system state information. As will be presented later, our work is different as, unlike many of the previous techniques, it considers the system state information and provides a generic abstraction framework. Our proposed method converts raw events to platform-dependent semantic events, extracts the system state value, and sends them as inputs to the pattern-matching algorithms.

In [39] the concept of trace abstraction aims to reduce the trace size, and the complexity of the data, especially for large data processing. In this same work, three data driven approaches for abstraction are presented: metric-driven, stateful data-driven and a structure-based. Moreover, they were used in [40] and [41]. While the first deals with kernel event generation, using a semantic approach rather than a low-level representation for modeling a state, the second paper deals with statistics in large trace files, using a statistics database of several metrics parameterised through a granularity degree.

Moreover, in this article, a metric-driven approach is explored, it is able to provide statistical data, demonstrating the issues related to this approach: first, the difficulty of efficiently computing the system metrics statistics without having to reread the trace events. The second is to find a way to support large traces. Although the use of those metrics can be difficult, this has been achieved in [85].

The approach of using an automata-based comparison, was proposed by [86], and aims to create a FSM that simulates the behavior of the system. In this work a pattern mechanism is applied and a list of problematic patterns is searched for using the State Machine Compiler language, SMC, for the description of problematic behavior. Once the state machine is built, the potentially huge amount of trace data is subsequently processed through it, to find the patterns and explore the false-true implications. In this article, the patterns focus on security evaluation and testing procedures.

The work of [132] introduces a declarative pattern specification language to find pattern-matching in kernel trace analysis. In this work, trace abstraction is explored to compact the information in the traces, summarizing the important data. For the specification of the language they used ANTLR.

However, the pattern matching technique, as explained above, is a limited manual analysis with linear comparison performance. Another problem is the false-true problems that might arise upon finding patterns in the traces.

Trace summarization is another possibility to analyse trace data, highlighting key aspects of large traces. The work of [66] proposed an approach for extracting summaries from large traces, relying on the removal of implementation details. The concept of trace summary is related to highlighting the main aspects of a trace, where the input to the method is a large

trace, and the summary will contain only the relevant aspects of it. This can later be used to represent the system using UML notation.
The methodology was applied in Weka [100] and was able to represent realistic aspects of this objected oriented library. The aspects for summarization are based on an empirical study, using a qualitative evaluation of the QNX software.
However, the problem with this technique, though, is the need to predefine the main aspects that need to be summarized in a trace. In this work, the key aspects of the summarization were compiled using a questionnaire.
Another methodology that can be used to compare traces is the analysis of the critical path of a task. In fact, a specific task within the trace is analysed, following its dependencies through the multiple related threads executions.
The critical path of an algorithm is a methodology to analyse the tasks that rely on multiple threads. For those problems, [50] proposed an algorithm that is able to find all the threads that contribute to the total time of an execution.
This algorithm efficiently retrieves all execution segments that contribute to the latency of a task. Using some event properties (the sched_wakeup) of the Linux kernel, it is possible to identify the multiple threads related to the specific task. However, this tool does not relate this information to user-space functions, which makes it difficult to use for studying application code. Furthermore, the interactive view is able to show only one execution at a time. For comparing groups of executions, this is fairly restrictive.

2.10.2 Automatic Analysis methods

For automatic analyses, there are several mechanisms. One of these techniques is the possibility to use machine learning and data mining techniques.
Spectroscope is another tool that uses statistics and high level abstraction to automate the analysis process, focusing on end-to-end tracing, as presented in [106] and [112]. This tool was designed to find changes in behaviour, not to find specific anomalies, and was used to find problems in two versions (or periods) of Google's Ursa Minor distributed software. Specifically for this software, five problems were described. It uses Startdust as end-to-end tracer. This adds some overhead on Ursa Minor performance, depending on the operation.
In its methodology, Spectroscope uses the Perl language and MATLAB statistical comparisons of normal and problematic periods. It used the DOT utility for plotting data, as often done in other similar tools.
The statistical test used is the Kolmogorov-Smirnov, a non-parametric test for mutation identification that compares the shapes and distribution of mathematical functions and uses

a ranking system for mutation identification. Spectroscope uses the normalized discounted cumulative gain (NDCG). For the performance evaluation, which has a range from 0.0 to 1.0. The Kolmogorov-Smirnov test is similar to the KW test, since both are non parametrics tests used for comparison evaluation.

2.11 Methods for analysis

We can divide the analysis techniques into two main categories: statistical analysis and data mining techniques. In terms of performance analysis, the use of machine learning and data mining techniques were already investigated in some interesting work.

2.11.1 Statistical methods

From the statistical point of view, several possibilities of parametric and nonparametric tests can be used such as the Confidence Interval, ANOVA and Kruskal Wallis tests. The first method, Confidence Interval, is a comparison of samples considering a percentage of tolerance. This can be seen as similar to a t-student test. This method transforms a relative value estimate into a scope of qualities that are thought to be acceptable for the population. The length of a confidence interval relies on its Standard Error(SE), which in turn depends upon two metrics: the sample estimate and the standard deviation.
In co-authored work, [1], we did an analysis on the confidence interval of several Chromium releases data to find significant performance regressions. The technique was able to find specific regressions in versions (version-specific data). This technique can be applied to more uses cases and can be extended to include other techniques, such as clustering the data in a preparatory phase.
The second method, ANOVA, can be defined as a way to determine whether there is any statistically significant difference among means of groups being compared. This technique is used in several areas of knowledge and has different approaches, including the MANOVA and Multi ANOVA.
Finally, the third method is Kruskall Wallis, which is also called ANOVA by rank. It compares medians instead of means as ANOVA. This method is a non-parametric method to compare groups, although it does not take in consideration most of the assumptions of the ANOVA method. In performance analysis, ANOVA was used to compare executions in [49]. In this work, the comparison of executions of Java garbage collection was studied, which has problems related to its different behaviour patterns. This problem is related in many aspects

to the JIT-process that influences the performance of the garbage collector. Although it is an empirical study, this work also focuses on the discussion of non-rigorous statistical deductions, which can occur in the analysis of similar problems.
Nonetheless, in our experiments, part of the executions did not generate a normal data distribution, and consequently required the application of a parametric method, such as ANOVA, for its analysis. This is a major drawback, that real data might face, and a reason for applying other methods such as Kruskal Wallis, which we explored in [88].
In [90], they used the Pinpoint tool to build runtime paths of applications, and to manage failures for large distributed systems. The tracing is done in the distributed black-box components service until completion, and is able to discover structural anomalies comparing the probabilistic context free grammar, PCFG, of the requests.The comparison mechanism uses ANOVA and other non-parametric comparison algorithms. The construction of those paths allows the comparison with normal and abnormal paths. It is interesting to highlight in this work that the solution is a combination of three parts, and tracing is just one part of it. Consequently, the solution could be used using other tracing schemes.
In terms of profiling-based comparisons, the work of [96] proposes a statistical approach for detecting performance regressions using control techniques. The approach works by building a linear equation which estimates the number of performance counter samples. The statistical approach is used to compare the predicted and the real counter values using Spearman correlation, and the Shapiro-Wilk test is used for the evaluation. The Shapiro-Wilk test is an hypothesis test that assumes as null-hypothesis that the population is normally distributed, as defined in [42].

2.11.2 Data analysis techniques

From the data mining aspect, which is about solving problems using the analysis of data already saved [133], several techniques can be used to find regressions and improve performance. Machine learning may be used to cluster and classify the data. The tools for this purpose include clustering techniques, decision algorithms and bayesian networks.
A common task in machine learning is to cluster the data, also called cluster analysis. It can be defined as separating the data in groups that might not be labeled. Support Vector Machine (SVM) is one algorithm used for this kind of task. In clustering algorithms, the output will be in the form of a diagram of clusters of data.
The first technique, SVM, is a classification tool in data mining algorithms, used to classify the data using hyperplanes. This algorithm requires a learning phase and might be applied to more than one dimension in terms of classification. Another technique is the hierarchical

classification algorithm, called agglomerative clustering, which clusters the data along a dimensional aspect.

This tool was used for performance analysis in [105] and [24]. The first work [105], called the Brain, is a dynamic analysis tool that clusters source code to investigate the behavioral aspects of the application. The mechanism was developed as a visualization tool for runtime interactions and was inspired by neural images. This work is related to Execution Murals, which is a way to display software runtime interactions.

In terms of visual cluster, the second work [24], created the Constellation visualization. Dynamic, run-time, information is used to augment the static program model. The tool is able to do clustering through a system dependence graph (SDG) at statement-level. The tool combines dynamic information from testing tools which impose a low runtime overhead. This work improves the visualization scalability of SDG, which are not often used, due to the poor scalability of the current tools.

Another tool, Magpie [6], defined as an automatic toolchain for analyzing system workloads, applies a behavioural clustering technique to system requests. It considers the resource consumption and other resources through a stochastic context-free grammar, SCFG. This tool works by analyzing the requests with state machines, and then building stochastic workloads that simulate real ones. The clustering part is done using a string-edit-distance metric in the data of the requests. This technique can also be used to detect anomalies, considering as well the causal ordering in the tool, which is used to produce a canonical annotation of each request.

Another task in machine learning is to classify the data, i.e. a function that classifies certain data considering its attributes. One way to do this is to use Bayesian models for the construction of this function, as a statistical modeling technique. The model basically works by building a set of conditional probabilities for each attribute, that will be subject to Bayes computation for the classification, as elucidated in [46].

The work of Cohen et al. [21] uses a supervised probabilistic model called Tree-Augmented Bayesian Networks (TAN), to identify correlations of system-level metrics related to high-level performance. This technique might be able to identify counters that are highly correlated with bad runs. Related to this technique, an interesting survey, compiling several of its aspects, was summarized in [10].

The third kind of algorithm, decision tree algorithm, is used to produce sequences of rules, or associations, which can be used to recognize the classes for decision making. They can work by ranking specifically a series of properties of a group, and how they impact the classification and are used in different approaches and contexts. The work in [138] proposed a technique comparing performance metrics, to diagnose failures in Internet sites. The approach was used

with traces from eBay, using algorithm C4.5, and produced two false positives but was able to find thirteen out of fourteen failures in a real use case. The decision tree was compared with an association rule algorithm, and was able to outperform it.

The table 2.1 summarizes the techniques used in several researches, while the table 2.2 provides a pros and cons analysis of each one.

Table 2.1 Techniques of data analysis applied for performance evaluation

Method	Intention	Used in
Decision Tree	Classification algorithm	[17]
Anova	Compare groups using mean	[18]
Non-parametric techniques	Compare Median or other metric	[88]
Apriori	Mine association rules	[3]
SVM	Divide the data using an hyperplane	[137]
k-means	Segregate the data in k clusters	[106]
Bayesian classification	Mine probabilistic rules	[73]

Table 2.2 Evaluation of each technique data analysis

Method	Pro	Cons
Decision Tree	More flexible than k-means	visualization tool
Anova	-More than one group -parametric classification	Assumptions that might not be applied
Non-parametric techniques	more flexible than parametric tests	non-parametric
Apriori	Find associations in any abstraction	Can make false associations
SVM	Optimized division of the data	Hyperplane
k-means	Optimal aggregation	K defined
Percentage classification	Can lead to just one group	Number of groups not defined
Bayesian classification	statistical classification	outlines might play strong role

2.12 Trace visualization tools

There are basically three main ways to visualize the traces, which will be presented below.

2.12.1 Gantt Diagrams

The first visualization tool for tracing and performance data is TraceCompass. It was developed by the DORSAL laboratory and Ericsson, in the Eclipse environment, and aims at the detailed analysis of several aspects of trace files. TraceCompass is composed of several sub-projects, for analysis and visualization, with specific characteristics.
TraceCompass supports several features, including critical path computation and call graph analysis. This can help the developer to highlight several aspects of traces by applying algorithms for comparison and analysis. However, this tool still requires deep knowledge of the system, as well as of the specific strategy to fix an issue.
Figure 2.1 is a sample view of the TraceCompass, [22]. This examples shows the Control Flow view, specifically for Python 3 profiling data.

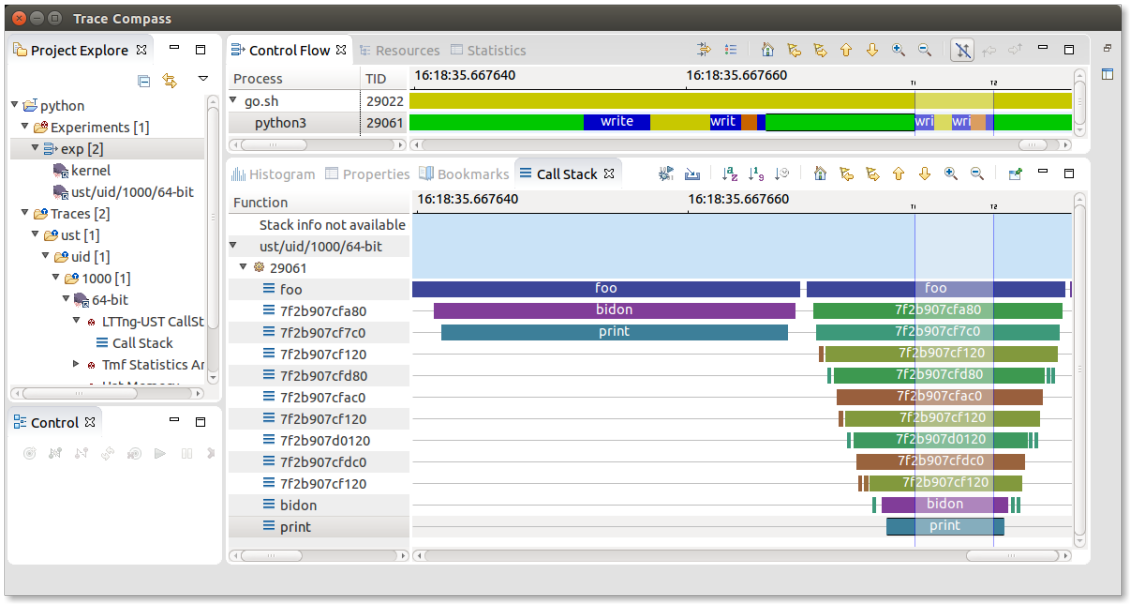

Figure 2.1 TraceCompass

Focusing on high-performance parallel computing libraries, the tool Paraver [15] is a visualization tool developed by the Barcelona Supercomputing Center. This tool currently traces MPI, OpenMP, pthreads, OmpSs and CUDA applications. It can be seen as a data browser for parallel traces. In this tool, the metrics are user-definable, computed from a series of functions and modules. The interesting aspect of these tools, besides the parallel traces analysis, is the possibility to use programmable definitions to gather than predefined metrics.

2.12.2 Flame Graphs

Another important visualization method, is the Flame Graphs, developed by [62], which are able to show cpu resources along with heat maps and icicle plots. Flame Graphs are not specifically a tool but rather a methodology for comparing differences on hot paths in CPU profiling. They can be generated in several ways and originally focused on CPU profiling through linux perf_events. They may be used to quickly find the functions that differ in terms of duration.

These tools were motivated by the limitations of comparing several CPU profiles, from normal tracers and profilers, where those tools could only display CPU profiling information with many lines of text. Flame Graphs can be used to solve problems such as comparing MySQL data and CPU profiling applications. However, Flame Graphs sometimes face issues related to incomplete stack traces, or missing function names. The incomplete stack traces may lead to incomplete conclusions.

A specific kind of Flame Graph, called Differential Flame Graph, DFGs, is able to compare two flames by using a Red and Green schemes. The original Differential Flame Graphs may have problems with the ambiguity of the color scheme, which has two colors only: Green and Red. This kind of flame graph is used specifically to highlight differences in two executions and find bottlenecks in functions on a program call stack.

This methodology requires the full stack trace, and consequently some tools might not provide all this information. The stack might also be incomplete or without the symbols loaded. In both cases, the stack will be partially reproduced.

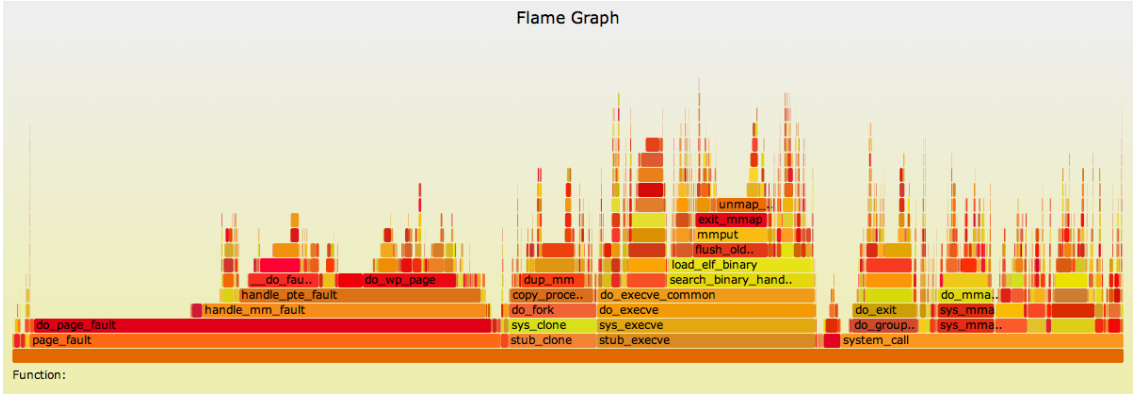

Figure 2.2 Flame Graph

The figure 2.2 is an example of CPU Flame Graph, [62]. In this example we verify that the flame graph is a visual representation of flames, i.e. profile data, in a bottom-up organization.

2.12.3 Graphs

In [2], a tool called Trevis, which is a visualization tool related to the calling context tree analysis, is presented. It is a visualization and also an analysis framework, with a different visualization pattern. It was developed to study the CCT produced from another tool called FlyBy software. Like TraceCompare, it relies on a calling context tree, CCT, about the caller-callee relationship.

This visualization and analysis framework includes several visualization tools such as radial display, TreeMapRenderer, linear and highrise renderer. It also includes a hierarchical clustering algorithm that uses the Lance-Williams dissimilarity update formula. This tool allows the users to play with the Calling Context Trees, and apply several methods to compare them. Trevis is able to compare the nodes using many metrics, such as Tree Edit Distance and Multiset Tree Distance. A dissimilarity matrix can be created to compare the similarities of the CCTs, and clustering methods can be applied.
However, this tool relies on human interaction and knowledge. Trevis relies on the data from the FlyBy profiler tool, to identify the slower executions. FlyBy provides thereafter a failure report, containing this information that can later be used in Trevis to be analyzed.

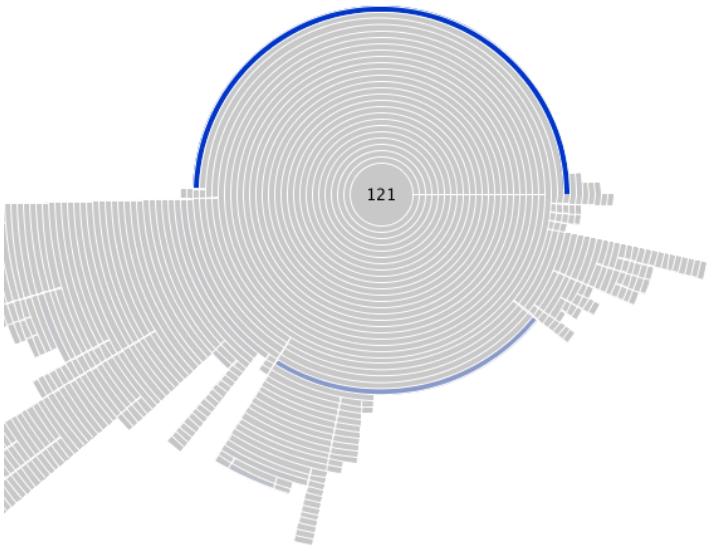

Figure 2.3 Example of Radial view in Trevis [Source [2]]

The figure 2.3 is an example of Trevis, [2] displaying a radial profiling data with 121 layers of depth. However, Trevis does not display the names of the functions in each layer.

2.13 Trace Correlation

Trace correlation has appeared in the literature, combining traces from userspace and kernel space. The work of Fournier et al. combines userspace and kernel space data, aiming to analyse complex programs in high-level languages. This require userspace tracing. User applications are often written in high-level languages but the availability of proper support for these languages is required. This strategy can be used to compare and find specific differences in the traces, aiming to extract more information than what can be produced with just one trace (i.e. kernel or user-space). This is interesting, since LTTng can indeed trace at both levels.
In the work of [68], a solution that does trace correlation using patterns is used to compare different versions of the same software. The traces from different versions are compared using a correlation metric, calculated in two approaches. First a non weighted approach is called NW_TCM and then a weighted approach is called W_TCM. The solution is composed of two phases, the pre-processing and the correlation calculation, using the already mentioned functions. The work is applied in the Weka [100] software, and was able to correlate versions 3.4 and 3.7, finding a 70 percent dissimilarity. However, this technique is not necessarily easily reproducible, since the results require a synchronization mechanism with userspace and kernel space traces
Finally, in [8], this technique was also explored for heterogeneous embedded systems, such as bare-metal CPUs. This work introduces bare-ctf, to generate LTTng compatible CTF traces on bare-metal systems, and uses traces synchronization techniques to correlate the traces from the different heterogeneous subsystems. Indeed, the solution is applied to the Adapteva Parallela Board, and represents a generic solution to bare-metal system tracing.

2.14 Trace Comparison

Several tools use the concept of trace comparison to obtain relevant information from the trace data, and to improve the software performance analysis in general. We highlight here three of those tools, specifically focusing on trace comparison.
One of those tools is TraceDiff[84], which does a comparison between traces by highlighting their differences. TraceDiff Compares two trace files and prints the details of packets that differ to standard output. This is useful for finding packets that are present in one trace but

not in the other, or for finding conversion or snapping errors. This tool relies on three main characteristics: scalability, robustness and ease of use.
However, as the other tools explained above, this tool requires the manual selection of the comparison traces and the visual analysis of their differences. The similar traces will be highlighted and connected, whereas the diferent ones will not be.
TraceCompare, was developed in the DORSAL laboratory, [30], and is a tool for performance comparison. It can be used in the two domains: userspace and kernel space. The methodology used in this tool is to create a tree data structure, a Calling Context Tree with metrics, from the tracing data, to measure the executions of a program. The segments are defined by the user as sequences of beginning and end, which will delimit the nodes in the tree data structure. The nodes also can aggregate performance data in those segments.
This tool was developed to compare traces of execution. It uses a javascript front-end and tibeebeetle [31] as back-end. To do this CPU profiling comparison, the GUI tools provide Differential flame graphs, similar to [61]. It was able to find abnormalities in the write function of MongoDB by studying the differences among several sample runs.
However, TraceCompare requires expert knowledge, even with a statistical metrics analysis implementation to help comprehension, as suggested in future work [28]. Also the comparison is restricted to two groups and the clustering selection is manual. This last characteristic gives an interesting opportunity for further tools, such as our proposed solution.

2.15 Root cause analysis and detection

In terms of performance, anomaly detection and bottleneck identification, PADBI, there is interesting ongoing research. Part of the research is relative to the detection methods, while the other parts concern the identification of root causes, [16].
Anomaly detection methods can be defined as part of monitoring and may or not use data mining techniques.
The work of [141] presents a Regression-based diagnostic framework for analyzing performance anomalies and potential causes of SLA violations in virtualized systems. Their approach is based on a variant of the Least Angle Regression (LAR) called Lasso, used to identify suspicious system metrics accounting for observed performance anomaly.
Another research, [135], models the relationship between application metrics and system metrics to explore properties of selection, reduction, and anomaly detection. The approach used combines the use of Fourier transforms, which will provide data to a statistical window-based average. The utility of the Fourier transform is to track patterns in the data. Several root causes analysis methodologies have been applied to this process. The term analysis is related

to the fact that those tools find the underlying cause of issues. Using kernel information, the work of [38] investigates causes of unexpected long delays in GUI-based user interaction applications. This work developed the collection of tools called TIPME, The Interactive Performance Monitoring Environment.
The tool CloudPD, from [115], is an example of a framework that uses linear correlation analysis to determine variations in Virtual Machines, with use cases like Hadoop and RUBiS. In this work there are several contributions, especially in terms of implementation. For root cause analysis, the use of correlation-based performance models can be highlighted, although a similar approach was used in [48].
The work of [83] detects anomalies in web applications using an approach called AOP-based monitoring, since it is based in Aspect Oriented Programming. It uses a correlation approach and time-series alignment to verify what is the root cause. The root cause analysis is performed through a module called Performance Analyzer. This model applies a computation procedure, and statistical correlation was done via three correlation methods: Pearson correlation, Kendall rank correlation and Spearman Correlation, where the Pearson correlation gave the best results, which requires an alignment using Dynamic-Time Warping algorithm, DTW.

2.16 Regressions Tests

Regression testing is performed after making a functional improvement or repair to a program. Its purpose is to determine whether the change has negatively impacted other aspects of the program, such as the performance, [94]. It can also be seen as a confidence test, proving that the new version works as well as the previous version.
Regression tests are important, since every change to the source code might degrade the software performance, by consuming more resources for example [98]. The importance of this kind of testing is the tendency to create new bugs in code throughout the development of updates. It is necessary to plan regression tests, before the software release, avoiding the deployment of buggy code to users.
In the area of releases comparison, the work of [93], which developed the tool PARCS, investigates regression tests using CCTs, using matching techniques and bytecode comparisons. The regressions are performed using the Apache Byte Code Engineering Library, BCEL.

Although not related to performance analysis, in [94] is addressed the problem of regression tests in objected-oriented code using a comparison of interprocedural control flow graphs,

ICFG. This paper concerns the test selection problem, specific to the C++ language, providing a code based technique using tests in the derived class.

According to the classification in [114], our solution combines ad-hoc, pair-wise, and model based analysis. First ad-hoc, since it deals with a specific number of metrics on the use cases. Secondly, pair-wise, since the comparisons are between groups with similar metrics, and third model based, since the clusters represent a model of the system to calculate the relations among the metrics. This is represented in Figure 2.4, [114], which summarizes the classification of several research papers.

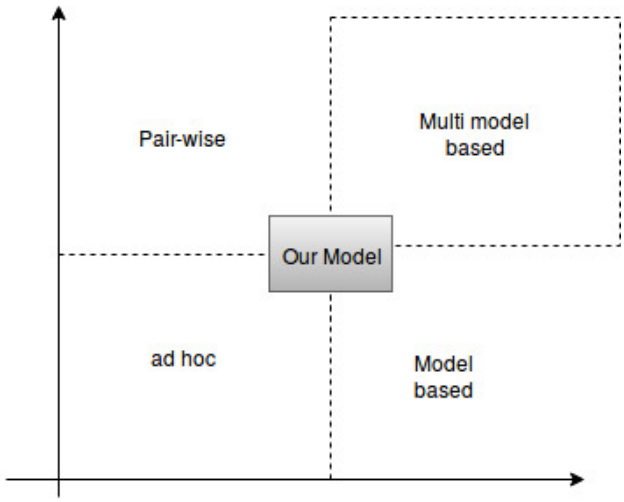

Figure 2.4 Classification of our solution according to [114]

2.17 Dynamic Data Structures

For dynamic analysis, several data structures have been used to represent the call-caller relationship. The profile data can be either used for code optimization or for general program understanding. The organization of the procedures and its order are usually represented in a call graph or a calling context tree. The call graph is a useful data representation for control and data flow programs, which investigates interprocedural communication (i.e., how procedures exchange information). It contains all the procedures and the relationships among the procedures in a program. A call graph can be represented with functions as nodes and caller-callee relationships as edges. Taking in consideration the context, i.e. the chain of methods up to the root, two main methods are used: Call Trees and CCTs.
In the Call Trees, there is an addition of a new node for each called method.

This data structure can represent with accuracy the complete chain of method calls in an application.

The other runtime structure, Calling Context Tree, CCT, provides flow sensitive profiling data within the nodes and was introduced in [4]. In this work, the hardware performance metrics from profiling were also taken into consideration. Calling contexts are very important for a wide range of applications such as profiling, debugging, and event logging. In [4], a call graph is used to compare executions and explore the so-called performance evolution blueprint. It is used to track the evolution of the system behaviour.

Following the previous work, and using tracing data, [31] aggregated performance metrics in a calling context tree to build an enhanced structure that could be compared. In this work, a tree with aggregated data, called ECCT, is built that relates the performance metrics to each node representing the functions of the application. This data is later analyzed offline and, by using the critical path of the task, is able to find differences in the metrics of the ECCTs.

In the tool Introperf, [76], the system stack traces are used to generate a specific CCT, which is called Performance Annotated CCT. In this tree, a comparison technique is used to compare the latencies. It was implemented using Windows ETW, developed by [89]. In terms of performance root cause, the paper explains the latency inference algorithm used for this calculation. They used this approach to avoid the requirement of source code or modification to the application.

Another tool that builds CCTs is PerfDiff, which targets VMs, [139]. In this work, a framework is used for comparing the characteristics of those trees. According to the authors, it is possible to compare the CCT's in two complementary ways: weight difference and topological difference. The first kind of comparison, weight difference, may be achieved in two ways: node based weight matching and sub-tree based weight matching. The topological differences are done via common tree matching, which is an algorithm they proposed and which states that nodes match between trees when all the nodes in the path from the root do match. This approach is applied to several Java and J2EE applications where the call counter is used as performance attribute.

To build the CCT, two main techniques were used previously to collect the stack frames of the applications, (in what is often called the stack walking process). Those are the exhaustive approach and the sampling based approach, although other methods such as static and adaptive bursting have been used.

When tracing all the function entries and exits, to build the CCT, the result will be the complete tree for the application. The overhead of tracing and processing all the entries and exits is a major drawback in this approach.

The second main approach is the sampling technique. This technique is able to reduce the overhead as compared to the exhaustive technique, but has two main disadvantages: the inference of calls, which may lead to incorrect information, and the sampling rate which may be insufficient.
Other approaches are possible such as static stack walking and adaptive stack walking. The same authors of PerfDiff previously proposed using the Adaptative Bursting technique for the stack walking process in [140]. This kind of stack mechanism avoids the overhead of creating the trees in a process of eliminating certain profile data redundancies.

In the article presented in the section 4, we also used a version of the frames with the metrics without the aggregation of ECCT. By analogy to the ECCT, the tree was named EDT, Enhanced Dynamic Tree. They are very similar, although the EDT is not summarized and can facilitate the mining process relating each node to the metrics without further computation.

2.18 Auto Grouping mechanism

During software execution, several parameters may change and the application performance can vary according to many factors. This can occur even when the algorithms used are deterministic and no probability is used. The cause for this variance is outside the scope of the program itself (e.g., operating system scheduler). However, some applications may have issues in the code, related to its implementation details. Software testing mechanisms might not be sufficient to reveal this behavior, especially in terms of performance discrepancies.
An example was found in [31]. After executing several times the same query operation on MongoDB, a free open-source database framework, its performance decreases abruptly. Further investigation lead to the root cause of this performance issue.
The auto grouping technique was presented as the solution for those kinds of problems. The approach is to do automatic comparison of different executions of the same program, under the same configuration settings, and investigating their metrics behavior.
The solution is automated by a heuristic comparison of several group distributions, and can be applied to an arbitrary number of groups, together with a clustering mechanism such as k-means. The automatic grouping gives the optimal number of groups in terms of SSE and, together with a comparative mechanism, can be used to find associations between groups. The division into groups enables the comparison and study of their variances using different approaches. For a small number of groups, the investigation by comparing the variances of the metrics of the groups can lead to the discovery of root causes. Another approach can be the use of the apriori algorithm, which can be applied to find associations among the groups. The complete solution is able to find associations between the performance and

metric variations, i.e. groups of different performances and metrics. This can indicate if a specific metric is responsible for a certain group of executions that have a bad performance, in a fuzzy approach, and that can be applied to infrequent problems.

2.19 Conclusion of the Literature Review

For the dynamic analysis of an application, tracers, profilers and debuggers can be used, each one with its own specific advantages and disadvantages. Profilers represent the caller-callee relationship and allow a deep understanding of the time consumed in an application. Profilers add extra overhead and may not be suitable to find performance issues that occur between groups of several executions of a software.
Debuggers can be used specifically to find a portion of code that may trigger issues during the execution. This tool relies on breakpoints that will stop the execution upon encountering a specific condition. This process allows specific conditions in the source code to be tested for bug finding.
Finally, tracers use advanced technologies and implementations to record events occurring at several levels in a computer system with minimal overhead. This feature makes tracing a useful mechanism to understand and analyze the behavior of the system under real load in the real production context.
By using tracing, it is possible to develop a data structure which represents the dynamic execution of a system but also can be embedded with performance data recorded within the tracing data. Several other techniques can relate this performance data with specific functions in the same node, and can associate metrics and functions. In summary, several strategies allow to extract, from one or more traces, all the events related to a given task execution. However, there is a demand for automated tools to analyze the performance.
This research is based on the several methods of comparing performance data that were introduced in this chapter, including data mining techniques and statistical evaluations generated using profiling and tracing mechanisms. In the next chapter, we present the methodology used throughout our solution.

2.20 Summary of the tools

We summarized the related work in two tables, Table 2.3 and Table 2.4, presenting several tools and their corresponding methodology.

Table 2.3 Comparing tools for performance analysis

Tool	Intention	Kind of tool
PerfDiff [[139]]	Generate CCT in VMs	Detection and analysis
TraceCompare [[31]]	Compare groups of executions	Detection and analysis
Spectroscope [[112]]	Distributed system behaviour categorization	Detection and Analysis
Introperf [[76]]	Generate CCT Based in ETW	Detection and analysis
Spectraperf [[9]]	Detection of I/O regressions	Detection and analysis
Pinpoint [[18]]	Build runtime paths on large systems	Detection and analysis
TraceCompass [Trace [22]]	Trace Analyze	Manual Analysis
Magpie [[6]]	Automatic tool chain builder	Detection and analysis
Trevis [[2]]	CCT analysis tool	Visual method Analysis

Table 2.4 Pros and cons of each tool

Tool	Pros	Cons
PerfDiff	Node and weight comparison	Topological differences might mislead
TraceCompare	Use directly in the browser	Manual select entries and exits
Spectroscope	Automated k-means	Analysis time and requires end-to-end instrumentation
Introperf	Integrated with ETW	-Complete stack walking -overhead
Spectraperf	Efficient performance Specific for I/O	Comparison of two software versions
Pinpoint	Comparison using a free context grammar	ANOVA requires normal distributed data
TraceCompass	Several internal tools and methods	Manual analysis
Magpie	Use of a comparative metric: string-edit-distance metric	Probabilistic construction of the event
Trevis	Manual method	Rely in another tool

CHAPTER 3 METHODOLOGY

In this chapter, the problem definition, research assumptions and research questions will be stated.

3.1 Problem Definition

The problem addressed in this work can be summarized as comparing several executions of the same software, measuring different performances aspects. Those aspects might indicate several factors, intrinsic and extrinsic, of the program performance.

3.2 Research assumptions

(i) It is possible to gather information from tracing records and understand it using high level abstractions and data structures.
(ii) It is possible to associate and/or correlate metrics gathered from traces, using mathematical techniques in a meaningful and automated way.
(iii) The differences among the groups will give an indication of the performance issues, i.e. association is often indicative of causality
It is also important to highlight that this work partially relies on previous work (i) [28].

3.3 Research Questions

Is it possible, using high level abstractions and dynamic data structures, to automatically find root causes for performance differences?
Minor question: What are the limitations of the current techniques and how to improve them?

Although currently performance analysis uses visualization tools as the main solution, quite often specialized knowledge is still required to interpret the information gathered. As a consequence, specialists knowledge is necessary. In real situations, however, time is a scarce resource and deep analyses may require too much time to solve problems in deployed systems, thus the motivation for an automatic approach in those cases.
Furthermore, the application of statistical comparisons may be limited in many cases, motivating the use of correlation techniques using higher level abstractions, e.g. ECCT or EDT.

3.4 Specific objectives

The aim of this research is to automate trace analysis, and several mechanisms are explored in this direction. From this objective, we were able to develop some helpful techniques to identify the root cause of performance bottlenecks in software applications.
The field of tracing tools is very specific, and relies on analyzing a considerable amount of information from the trace files. In fact, just a few seconds of recorded traces can generate an enormous amount of information to be analyzed. Therefore, there is a demand for tools capable of quickly analyzing this information. However, at the same time, there is a problem with current tools, since the interpretation of this data depends on specialized performance analysts. Specifically we want to compare the execution of the same software to analyze its behavior, by categorizing them in groups and analyzing each group in terms of performance metrics, i.e. cache misses and page-faults.

3.5 Solution Design

The design of the solution was developed to target two specific aspects: automation and flexibility. The first aspect, automation, can be outlined as the need for the solution to be used with minimal intervention. That is, it can be used without a specific preparation phase of learning and labeling the data. Moreover, the presented solution should not require deep statistical knowledge and is complementary to some clustering techniques such as k-means, which require the number k, i.e. the number of groups.
The second aspect, flexibility, implies that it can be used with different grouping and clustering techniques. The heuristic comparison should allow the use of many clustering and grouping mechanisms, such as the k-means algorithm.
As a trade-off of the approach, the computation time may be significant, since the heuristic comparison applies several times the execution of clustering methods.

3.6 Approach

The approach used for the solution can be divided into two parts: the data collection tools, pre-processing and the data analysis methods, presented below respectively.

The process of analysis of the data is explained in the diagram 3.1.

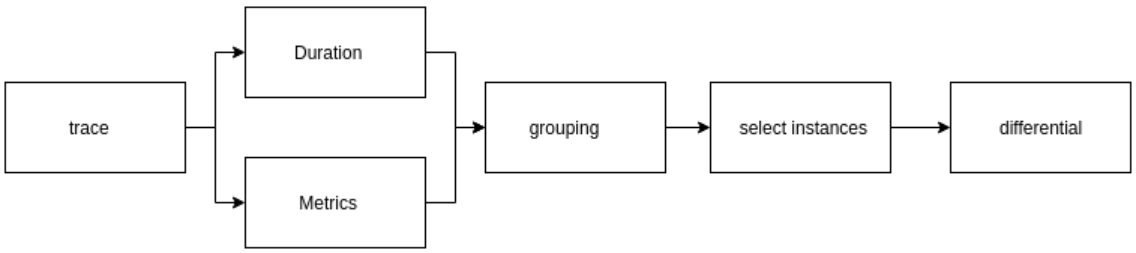

Figure 3.1 Process Diagram

3.6.1 Data collection tools

The tools used for data collection were LTTng and Perf. The applications were statically instrumented and compiled with the instrumentation. LTTng induces a small overhead, so the traced application can run with a negligible performance difference, as compared to the normal run. Perf is able to provide the hardware and software performance metrics (performance counters) for the executions.
The tracing of the executions provides the data that will later be used for the construction of the tree, (whether a summarized tree, Enhanced Calling Context Tree, or not summarized tree, EDT, Enhanced Dynamic Tree, is used).

3.6.2 Pre-processing

The information in each node is read and an auxiliary data structure is used to be able to do the comparisons. With the one or several metrics in each node, a table or array of execution metrics is built.

The auxiliary data structure is partially presented in the Table 3.1 and aims to transform raw data into a ready to process format. Later queries can be executed to read the one or several metrics at the same time.

Table 3.1 Auxiliary Structure

Execution	Metric1	Metric2	Metric3
1	10	40	50
2	40	40	50
3	30	40	50

3.6.3 Data analysis methods

In terms of data analysis, several methods were applied, for example: k-means, percentage classification and Support Vector Machine. Later, the auto grouping technique was applied to automate classification. This research focuses on the so called hard-clustering, which is different from fuzzy clustering algorithms, such as EM and Gaussian Mixture Model. Each technique is described below:

The first technique, Support Vector Machine, is a grouping method based on a hyperplane, splitting the data in two parts. To use this technique, a model needs to be trained according to a training data set. The data is recorded, pre-processed and the hyperplane will later be defined to divide the data in two sets, e.g. the fastest and the slowest executions. Those two groups can be used to find associations between each group and their metrics. The metrics are sorted, and the method is applied several times to do the classification.

The second method, k-means, is a centroid algorithm which does optimal data grouping. This algorithm guarantees the best data partitioning for a specified value of k. The value of k is the number of groups and must be given a priori. The analysis is later performed comparing the statistics of each group, especially the fastest and the slowest. For simplicity, in most use cases we compared just two groups and found associations between a specific metric and its behavior. For example, the slowest group in a server application had considerably more cache misses than the fastest.

The auto grouping technique, using heuristic classification, was later used to fulfill the need for an automated flexible classification mechanism. The mechanism is based on comparing a statistical measurement of deviations within the groups called SSE. This technique is able to find groups automatically since the SSE distribution, which is related to the internal variance of the groups distribution, and its patterns represent a curve which can be used to find an optimal value, the so called elbow heuristic. This enables the comparison of several SSE of different groups and can be used without a pre-classification phase.

3.6.4 Testing Framework

We developed a testing framework in C++ using the QtCreator Framework. The framework was used to test the concepts of the models and grouping techniques by simulating a series of specific performance issues that are measured using perf counters.

In those specific scenarios, such as high cache-misses or page-faults, the measurements were made in such a way that they are later exported to a comma-separated-value, CSV, file.

3.7 Solution Application

The solution can be applied to several scenarios where the performance counters are able to measure the real behavior of the system, and tracing has a minimal or negligible influence on the real program performance.
An effective application of the solution are regression tests, as described in the literature review, targeted at finding performance regressions from one version to another. Indeed, the solution (or at least part thereof) can be used in those cases.
The solution can also be fully or partially integrated into a testing framework, aiming to find the performance issues before the deployment of the application.

3.8 Co-authored articles

In partnership with colleagues [1], we did performance measurements using Confidence Intervals. This work was accepted at the IEEE International Conference on Software Quality, Reliability and Security, QRS 2017. We applied a comparative technique to data from Google Chrome performance measurements. The challenge was to detect performance deviations among Chromium software releases. The method used was to compare the confidence intervals, using the median instead of the mean, as a way to compare several releases and identify statistically significant performance variations.
As result, we were able to demonstrate that this method of confidence intervals, using medians, is able to compare the releases, similarly to a t-test, and was able to determine performance deviations, at a function-level granularity, with a small number of trace samples.
The method showed a better performance than the standard Confidence Interval, which only considers the average. This method can be combined with a clustering technique to avoid (or at least reduce) the influence of outliers, which can make difficult the analysis of comparative data. Another possibility of using clusters is to enhance the confidence of the comparative intervals.

3.9 Authored articles

Chapter 4 will explain in details the specificities of the article. This article presents our proposed solution to find the root cause of performance problems using automated analysis.

The first step is to trace the application, recording the performance counters. The experiments were conducted using LTTng and Perf. The second step is to execute the program several times, until enough data about performance variations was found. The data is then recorded and a dynamic data structure, with the tracing data and the performance counters, is constructed. Then, the third step is to run the analysis mechanism to group the data automatically. After the groups are segregated, it is possible to compare their inner characteristics, i.e. the metrics recorded in the traces. A statistical approach can be used to compare the groups directly, but also a frequent mining association, such as the Apriori methodology, can be used to associate the groups as well.
The different tests show the flexibility of the model, which can be used in combination with a clustering or grouping mechanism for the grouping division.

CHAPTER 4 ARTICLE 1: PERFORMANCE ANALYSIS USING AUTOMATIC GROUPING

Authors
Isnaldo Francisco de Melo Junior
École Polytechnique de Montréal
Isnaldo-francisco.de-melo-junior@polymtl.ca

Michel Dagenais
École Polytechnique de Montréal
michel.dagenais@polymtl.ca

Index terms – Performance analysis, Tracing, Software visualization, Heuristic, Groups.

Submitted to Software Practice and Experience.

4.1 Abstract

Performance has become an important and difficult issue for software development and maintenance on increasingly parallel systems. To address this concern, teams of developers use tracing tools to improve the performance, or track performance related bugs. In this work, we developed an automated technique to find the root cause of performance issues, which does not require deep knowledge of the system. This approach is capable of highlighting the performance cause, using a comparative methodology on slow and fast execution runs. We applied the solution on some use cases and were able to find the specific cause of issues. Furthermore, we implemented the solution in a framework to help developers working with similar problems, along with a differential flame graph viewing tool.

4.2 Introduction

Software performance is a major concern for software development. Various studies highlight the use of tools such as debugging and tracing to help with performance problems. These tools can be used to improve performance or detect performance issues. One example of performance issues is the comparison of similar executions of the same program, in the

same configuration setting. An example was found in [28]. After executing several times the same query operation on MongoDB, a free open-source database framework, the performance decreased abruptly. A further investigation lead to the root cause of this performance issue.

For this kind of performance problem, which is related to the execution inside a real system, the current solutions are to debug or to trace the program. The first one, debugging, is to locate the code and problematic executions, directly in the source code, by executing the code in a test environment using breakpoints. However, some debugging tools require the reproduction of the exact issue to trigger the same code mechanisms, while it is necessary to stop (and thus delay) the execution of the program while debugging.

On the other hand, tracing generates an execution log of a software, that consists essentially of an ordered list of events. An event is generated when a certain code path is executed, the location being called a tracepoint. Each event consists of a timestamp, a type and some arbitrary payload [51]. Tracepoints can be embedded in the code in two ways: statically or dynamically inserted. The latter, dynamic tracing, enables the possibility to add tracepoints without modifying the source code, and the former requires the modification of the source code and subsequent recompilation. Besides, tracing can be performed at the kernel and at the user-space level [51].

Unlike debugging, it is possible to trace a program without interrupt it. Yet, there is some overhead caused by its usage. A good tracer needs to minimize the disturbance of the running program to be a useful tool for analysis. LTTng[26], developed by Mathieu Desnoyers, has this minimal level of impact on the system, and consequently allows to trace the user-space and the kernel space with minimal interference. LTTng, allows the analysis of task interactions, with each other and with the operating system. Locating and analyzing performance problems is not a trivial activity, because of the potentially large trace size, since more events generate more information to be gathered and analyzed.

After collecting the data with the tracer, it is necessary to analyze the software behaviour through some mechanism, for instance the call graph [110]. A call graph is a representation of the stack frames of the software and can be built using different techniques. This analysis requires expert knowledge and deep analysis of the system, since this process was not automated yet.

Through tracing mechanisms, it is possible to build a dynamic model of the software, for instance a call tree. Moreover, tracing allows the possibility to add performance measurements to this structure, as shown in [31].

In summary, from the enhanced data structure described above, and considering the lack of an automated solution to solve problems as the stated above, it is possible to build a solution that records several software properties at run time. Moreover, using some specific grouping mechanisms, it is possible to find root causes of several performance issues using a comparative approach.

This paper introduces an automated solution for clustering metrics, using the call context tree, to find performance related issues. We implemented it within Trace Compass, with visualization mechanisms to facilitate the analysis of different aspects of the Calling Context Tree. Then, we applied this technique to four different use cases, to analyze their performance problems. Finally, we discuss the drawbacks of this technique and the possible solutions to overcome them, aiming to apply this analysis to complex software systems.

The research aims to investigate the following research questions;

- RQ 1: How can we build an efficient and flexible model for performance comparison?
- RQ 2: How to automate the performance analysis on several runs using performance counters?
- RQ 3: How accurate are the obtained results ?

This paper is organized as follows The related work is presented in section 4.3, then the solution is presented in 4.5 with the clustering technique. In section 4.6, we present the methodology used followed by the micro-benchmarks and an illustrative example of the analysis. The use case section 4.9, the discussion of the results 4.10, a section about some limitations 4.11 and finally the conclusion 4.13 follow.

4.3 Related Work

In this section will be presented the basic principles of current tools used to find performance issues. The related work was divided into: Data collection, Analysis tools and Visualization tools, as described below.

From the perspective of Data Collection, there are two main tools related to this work, LTTng and Linux Perf Events.

The first, LTTng, Linux Trace Toolkit Next Generation [26], is a tracer that can record events from the Linux kernel and from user space applications into a single trace [29]. It is also

designed to have a minimal overhead on traced systems. It is therefore well suited to our goal of collecting all the factors that contribute to the execution time of tasks in a production environment.

The second, Linux Perf Events, is a profiler tool for Linux 2.6+ based systems that abstracts away CPU hardware differences in Linux performance measurements and presents a simple command-line interface. Perf is based on the perf events interface, exported by recent versions of the Linux kernel. This article demonstrates the perf tool through sample runs [74]. It is possible to record both software and hardware events [129].
It is interesting to highlight that the Perf tool can be used to record profiles on a per-thread, per-process and per-cpu basis [91].

There are two main Analysis tools related to this work, TraceCompare and Trevis.
The first tool, TraceCompare, was developed in the DORSAL laboratory [30]. It creates an enhanced Calling Context Tree to measure the metrics from specific segments of a trace. Those segments are defined by the user as sequences of Begin and End. This tool was developed to compare traces of executions and it uses a javascript front-end and the tibeebeetles library [29] as back-end. To do this CPU profiling comparison, the GUI tools provide Differential flame graphs, [58]. It was able to find problems in the write function of MongoDB after several runs. However, TraceCompare requires expert knowledge and also some statically significant metrics for the analysis [28].

The second tool, from Lugano University, Trevis [2], is a visualization and analysis framework. It was developed to study the CCT produced by another tool called FlyBy. Like TraceCompare, it relies on a calling context tree, CCT, on the caller-callee relationship. Trevis is a visualization and analysis framework that allows the users to play with the CCTs by applying several methods. However, this tool relies on human interaction, which occurs at the stage of FlyBy, to label the slower executions. FlyBy provides thereafter a failure report, containing this information that can later be used in Trevis to be analyzed.

A third tool is Spectroscope, which uses statistical and high level analysis. It was in fact designed to find changes in behaviour, not specific anomalies, and it was used to find problems in two versions (or periods) of Google Ursa Minor distributed software. Specifically for this software, five problems are described. It uses Startdust as end-to-end tracer and it added some overhead on Ursa Minor performance, depending on the operation.

It uses the Perl language, and MATLAB for the statistical comparison of normal and problematic periods. DOT is used for plotting visualization graphs. The statical comparison used is the Kolomogrov-Smirnov test, which is a non-parametric test for mutation identification that compares the shapes and distribution of mathematical functions and later uses a ranking system for mutation identification.
Spectroscope uses the Normalized Discounted Cumulative Gain (NDCG) for the performance evaluation, which is a range from 0.0 to 1.0. Spectroscope is similar to Pip and TAU [108] [111].

Finally, Introperf is a tool that uses system stack traces to generate a Performance Annotated CCT, called PA-CCT. Then, it ranks the latencies and compares them [76]. The intent of this tool is to be used in a post-development stage. It was implemented using Windows ETW[89]. The article explains the latency inference algorithm used for this calculation. They used this approach to avoid the requirement of source code availability, or application modification.

To visualize the output of comparison techniques, the flame graphs and differential flame graphs, developed by Brendan Gregg, are very popular techniques. As defined by Brendan Gregg, [58], Differential Flame Graphs can be a useful tool for comparing executions. A Differential Flame Graph is a visualization technique highlighting the differences between collections of stack traces (aka call stacks), from two different executions. Flame graphs are commonly used to visualize CPU profiler output, where stack traces are collected using sampling.

The use cases proposed in this work are related to regression analysis, where the performance of a software application decreases when comparing a new version to an older one. Similar cases were studied by [114] and [97], which focus on use cases related to the software Dell DVD Store. The first work used a hierarchical clustering approach, while the second work used a control chart approach.

4.4 Motivation

The main motivation for this research was the current limitations of the current tools to quickly track problems on executions that do not appear often, for example one longer execution run among a thousand that perform correctly. The difficult lies mainly in the amount of data that needs to be generated and analyzed comparatively.

From the point of view of the grouping techniques, the current more reliable technique still

requires some human analysis of the data, which consumes time and prevents automation. This motivates the development of the auto grouping technique, which combines an heuristic evaluation.

We also proposed and implemented the RGG differential flamegraph, to compare groups of executions. This implementation uses three colors: red, green and gray. This was necessary to avoid the ambiguity in the original work developed by Brendan Gregg, where the green color could mean faster execution or equal execution, during comparisons.

This methodology can be applied to other scenarios and is independent from the implementation. Besides, it is also independent from the grouping algorithm, since it is an heuristic and not an algorithm. Combined with the Apriori algorithm, the grouping technique can provide strong insights into complex cases.

We implemented the solution within Trace Compass, because of its flexibility and reliability as trace analysis framework. TraceCompass is currently supported by a large open source community, with many core developers from Ericsson, and the DORSAL laboratory at Polytechnique Montreal. It offers several features, which together form a complete analysis framework for different kinds of traces.

4.5 Solution

In this section, we discuss the solution developed, starting with the data structures, followed by the grouping mechanism algorithms, and then the clustering algorithms and the overall methodology.

The methodology can be summarized as follows:
First, we trace a program using statically or dynamically embedded tracepoints. Then, we read the trace and build a CCT record, along with the performance metrics. Then, we run the clustering techniques and the association rules, which indicate the possible cause for any performance issue.

Recording of execution

We record the program execution using LTTng, a low overhead tracer especially suitable for this type of research. The trace is also recorded with performance metrics such as instructions, cache-misses, page-faults, and scheduling switches by using the perf counters tools in Linux.

Generating the data structure

After recording traces for different scenarios, a pre-processing step builds the structures required for comparisons. This structure is an enhanced calling context tree. For this process, we need to divide the traces in segments corresponding to different instances to compare. For example, to compare different instances of file openings, the systemCallOpen and systemCallExit may be used as delimiters. In this process we aim to construct comparable information using ECCT (or EDT), where each node represents a call, and the information and metrics associated with this call are stored within the nodes. A delta of the entry and the exit for each metric is recorded in the node.

4.5.1 Pre-analysis Phase

The metrics from each node in the tree are placed into an auxiliary table data structure. The hierarchical tree structure is used for display purposes (e.g. Flame Graph) but not for the clustering phase. The automatic grouping techniques were applied for each metric separately. In the future, multi-dimensional clustering will be investigated.

Clustering technique

After the data is pre-processed, the tree is built and the auxiliary data structure is constructed, a mathematical model is constructed and its behaviour, compared to its inputs, can be determined to cluster instances into groups.

Execution Comparison

With the groups formed, the next phase is to compare them together. There are different ways to compare the groups: compare the mean and median for each group, or use the Apriori Association described in section 4.5.6.

4.5.2 Data Structure

Calling Context Tree
Calling contexts are very important for a wide range of applications such as profiling, debugging, and event logging. Most applications perform expensive *stack walking* to recover contexts[124]. The resulting contexts are often explicitly represented as a relatively bulky sequence of call sites. The goal of calling context encoding is to uniquely represent the current context of any execution point using a small number of integer identifiers (IDs). This

data structure was introduced by [47] and reused by [79], [134]. In this work we aggregate data, related to the performance, in the Calling Context Tree, which brings the concept of enhanced structure. The aggregated data is related to the performance metrics. The data is added to the tree nodes and enables off-line analysis. Figure 4.1 demonstrates the difference between the dynamic tree and the calling context tree. The metrics recorded inside the tree are described in the subsection Performance Metrics.

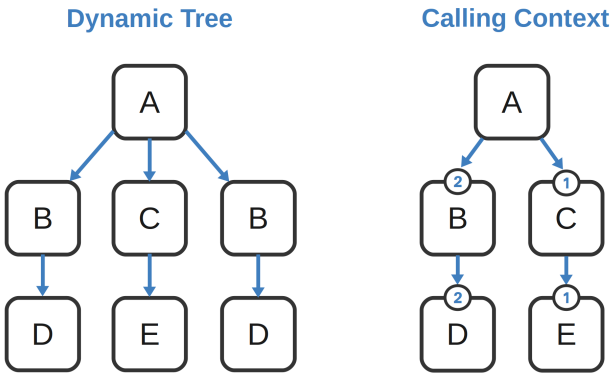

Figure 4.1 Dynamic Call Graph vs Enhanced Calling Context Tree

Performance Metrics
Because we generate the tree through a tracing approach, it is possible to record runtime information about the system [51]. It is recorded using the lttng feature add-context, which gives the possibility to add performance counters samples in the trace session.
Example: perf:cache-misses, perf:major-faults, perf:branch-load-misses
This technique was explored in the work of [28] and [101].

4.5.3 Data Structure Construction

The construction of the ECCT involves reading the trace and simultaneously building the nodes, as the trace data is processed in order. It is necessary to delimit the boundaries of the nodes in the tree. Therefore, specific events must be set as Start and End points of the nodes. Depending on the case, it may be easier to construct a simpler ECT, instead of ECCT.
Consequently, it is necessary to demultiplex the events in the trace. To do so, the trace must provide a way to identify the start and end points of each execution instance. For this process there are two approaches:
The first is to use existing events from the Linux kernel. As an example, the syscall_exit_accept event (generated when a connection is accepted on a socket) and the syscall_entry_shutdown event (generated when a connection is closed) correctly delimit requests received by an Apache

server.

The second is to use LTTng-UST probes, statically inserted in the source code. Different probe types can be used to delimit different execution types. In that way, the delimitation of the nodes can be achieved. The advantage the first approach is to use existing events, and thus no access to the source code is required. The advantages of the second is that no kernel knowledge is required to use this process.

Figure 4.2 demonstrates the mechanism used to create the ECCT from the trace file.

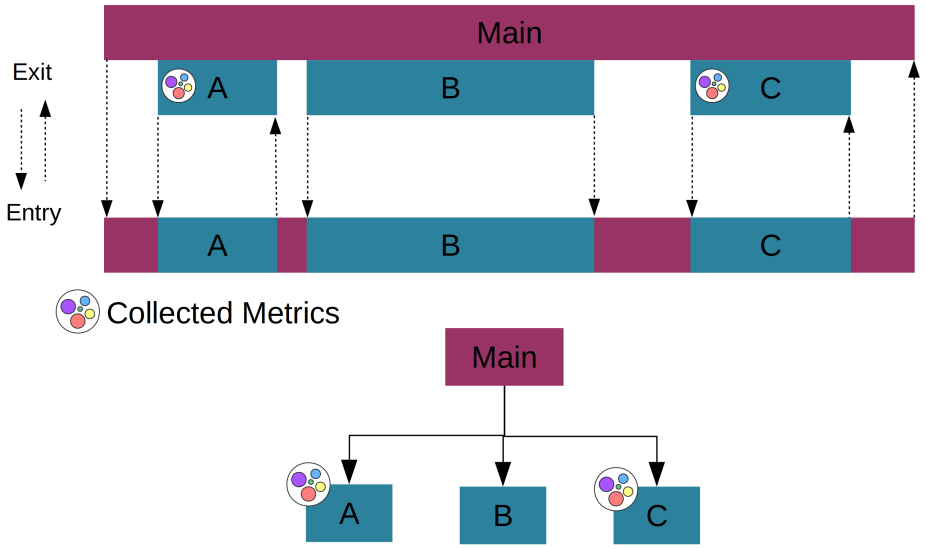

Figure 4.2 Enhanced Calling Context

After the construction of the tree, the clustering mechanisms, described below, can be applied.

4.5.4 Classification Strategies

In this part, we outline the clustering techniques used for the metrics grouping, the heuristics for the automated classification mechanisms, and finally the association rules used to find the metric which impacted most the performance of the executions.

Support Vector Machines

In machine learning, Support Vector Machines (SVM) are supervised learning models, with associated learning algorithms, that analyze data used for classification and regression analysis. Given a set of training examples, each marked as belonging to one or the other of

two categories, an SVM training algorithm builds a model that assigns new examples to one category or the other, making it a non-probabilistic binary linear classifier. Besides the requirement of teaching the model, the restriction to binary classification is another drawback of using SVMs. Indeed, with the division into two groups, relevant information could be lost and no further comparison techniques could be applied later.

Percentage classification

A different strategy, related to 1D classification of data is the percentage classification. This strategy is achieved by comparing the metrics and separating them by a percentage threshold, which is based on the mean of the group. The percentage classification is interesting, considering that sometimes the distribution is not well laid out to create clusters. The naive classification will draw a group, even when they are too close and could be in just one group with other clustering techniques.

K-means

K-means is a simple unsupervised machine learning algorithm that groups a dataset into a user-specified number (k) of clusters. The algorithm is somewhat naive in forcing the data into k clusters, even if k is not the right number of clusters to use. Therefore, when using k-means clustering, the users need some way to determine whether they are using the right number of clusters. Since the number of groups must be known before applying the k-means algorithm, another technique needs to be applied in order to find the appropriate number of groups (k).

Comparing Models

Comparing the three techniques described above, the conclusions were as follows. First, the SVM model was able to delimit the differences between the slow and fast executions. However, the delimitation is restricted to just two groups.

The second mechanism, Percentage classification, is an unsupervised mechanism and leads to the segregation of data, even if they are homogeneously distributed among the dataset.

Finally the k-means algorithm is efficient but requires the number of groups to be used in the classification process. Comparing the models, we applied several techniques including an heuristic classification to make the clustering process automatic.

Auto Clustering

Considering the models shown above, we chose to develop a non-supervised method, called auto clustering. The possibility to use an automated approach is more interesting for us, in comparison with non-automated methods, mainly because we aim not to use data to train the model. Therefore, we implemented a version of comparative k-means using the SSE (sum of square errors) variability information, plus a heuristic evaluation. This technique can be

used for an arbitrary number of dimensions, since the amount of difference, SSE (sum of squared error), can be calculated in these cases [107].

4.5.5 Automatic Clustering through heuristic Evaluation

Elbow method

One method to quantitatively measure the number of clusters is the elbow method. This method compares the sum of squared errors (SSE), considering several numbers of groups from the classification used. The elbow method gives the possibility to use the SSE to find the elbow value, which can be defined as a value at which the SSE changes its behaviour abruptly. In our cases, the elbow value is when the SSE stop decreasing substantially.

However, the elbow method does not guarantee a perfect match in cases where the data is well distributed. Instead, the analysis of the SSE can give a smooth curve and the best value for the number of groups is not precisely defined. In cases like this, we developed another clustering based on the mean distance of the data.

Heuristic Evaluation

To compare the SSE values, we needed also to do a heuristic function which compares the different values of the SSE, to compute the Elbow. Therefore, we use this approach to compare several runs of classifications and extract the one with the smallest squared errors. The heuristic used is to take as optimal group the biggest gap in an array of SSE values.

Figure 4.3 shows an illustration of the SSE and the elbow value. The elbow value is the number that marks the change in the path of the function. In our case the number of groups is 2.

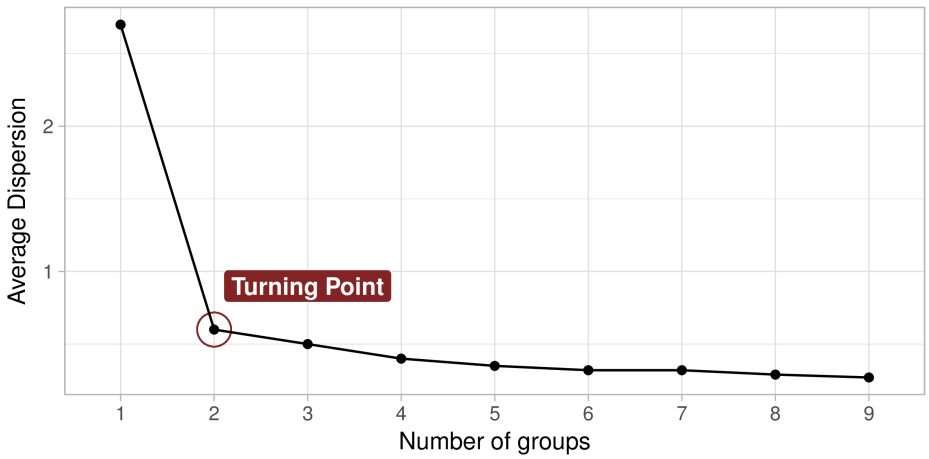

Figure 4.3 Elbow method: SSE Comparison

Figure 4.4 shows the SSE differences considering several number of groups. This image clearly shows the biggest gap between one and two groups, but since we are assuming that the data has gaps, we are not using one as the optimal number.

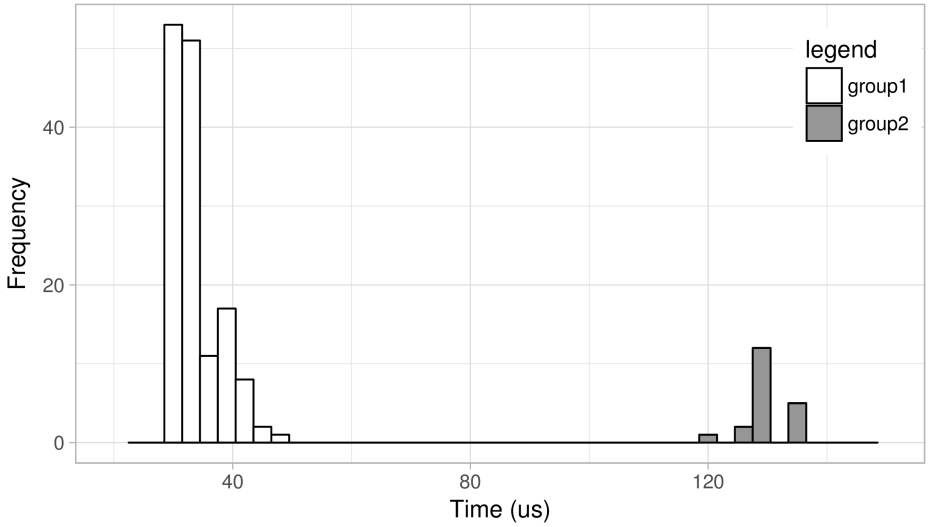

Figure 4.4 Automated clustering of the executions into 2 groups

4.5.6 Association among the Groups

The clustering of metrics is just one part of the approach, since a rule of groups needs to be applied to find the specific cause for the discrepancy between the executions. To solve this problem, and find the cause of the difference, we applied a set association rule after the grouping mechanism. Therefore, using a set exclusion, we can find the metric that is responsible for the elapsed time. The association rule is illustrated in Figure 4.1, which describes a metric X and the elapsed time comparison. The grouping on Metric x divides the data into two groups, and those groups are intrinsically related to the elapsed time group. The association rule can be applied in an arbitrary classification algorithm with several different dimensions. Thus, the association can be defined as a heuristic to find the root cause of problems, using grouping or clustering algorithms.

A matrix of groups correlation can be constructed to better understand the relations among the groups.

4.5.7 Accuracy of the model

The association will identify the group of metrics that are related with slow and fast runs. However, there is a possibility of false positives and false negatives. The accuracy of the

Table 4.1 Association of groups through Apriori algorithm

	A	B	C
A	X	75%	100%
B	75%	X	65%
C	100%	65%	X

model is related to the size of the groups, i.e. that all slow executions will be in one group, even though the related metric identified as the reason covers more runs than the associated group. In summary, if the groups overlap, range of values for the main metric and slow executions, no false positives of negatives were found. However, the correlation (overlap) does not mean causality for the performance problem, it is only an indication factor. In this matter, statistics are an indicator of underlying causes, which require some complementary analysis to be confirmed.

4.6 Solution Implementation

We describe in this section the implementation of the Calling Context Tree in Trace Compass, the Flame Graph comparison, and the Auto cluster mechanism.

The CCT View is an analysis developed in the Trace [22], which specifically aims to study the CCT and its variations. The module with Auto clustering was included in it. The view displays the calling functions in a graphical view that helps the developer/tester to improve their code, and to compare the traces using the approach described in this work.

Tree Construction

The CCT View builds the tree from the trace as explained in Section Tree Construction. In summary, the trace is read sequentially and event entry-exit pairs are grouped into nodes, and sub-nodes defined by pairs of entry-exit nested inside nodes. The data found, i.e the performance metrics, for the nodes is summarized for the respective node. The tree is build to summarize the redundant nodes, thus avoiding to build a Dynamic Call Tree. Figure 4.5, shows this feature in the CCT View.

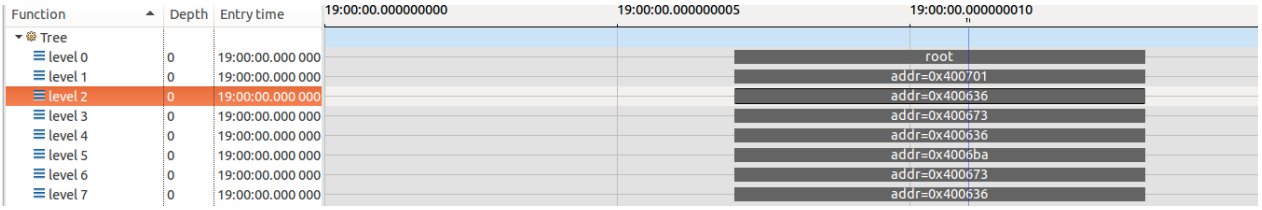

Figure 4.5 CCT View in TraceCompass

Differential Flame Graph

The CCT View implements a Differential Flame Graph view, to compare executions and groups of executions. Usually, the Differential Flame Graph compares two executions, but in our implementation it is possible to compare groups of executions. Our flame graphs are composed of three colors: green (to represent equal amount of time), red (slower executions) and gray (faster executions).

Figure 4.6 shows a diagram of the RGG Differential Flame Graph, the green color shows the functions which are faster in the main execution, the grey part means they are equal, and the red part means that this function is slower than its counterpart in the comparison group. The name, RGG flamegraph, alludes to this concept, in contrast with Brendan Gregg's Red/Blue Flame Graph.

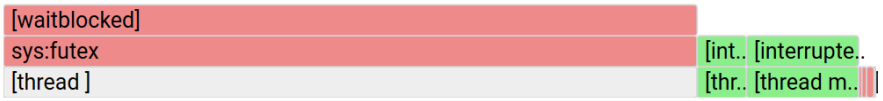

Figure 4.6 RGG Differential Flame Graph Diagram

Auto cluster

The Auto cluster feature was implemented in the CCTView to help developers analyze the cause for performance variations from traces. The heuristic evaluation (elbow method) and the clustering techniques (k-means and percentage clustering) were implemented separately, therefore the developer can choose the most suitable one.

Correlation feature

Another feature of the CCT View is the possibility to find the correlation matrix of the metrics. As the dictum states, *correlation does not imply causation*, which means that correlation

by itself cannot be used to infer a causal relationship between the studied variables.

4.7 Benchmarks

Considering the several aspects of software performance, we have highlighted some micro-benchmarks which can be sentitive to the performance of the software, and the performance could change throughout the releases.

Inline
The inline functions are a compilation optimisation, available in C++, to reduce the execution time of a program. Functions can be compiled as inline automatically, or can be declared by the programmer as inline functions. The compiler replaces the definition of inline functions at compile time, instead of calling the function definition at runtime, thus saving the execution of call and return instructions.

However, those functions do not always improve the performance and can deteriorate the performance [70]. To evaluate this claim, we tested this feature with several possibilities: strings, integers, structs, vectors, and classes. Using the inline mode improved the performance for all of them, except for std::string operations.

The reason for this difference is possibly related to the dynamic allocation time of strings in C++, as explored by [64]. We did several tests with different data structures and objects, and strings seems to be the most affected by this.

After investigation, we came to the conclusion that the root cause of this performance degradation is the influence of cache misses in the operations with strings inside inline functions.

The results of the benchmarks comparing the performance of inline functions and regular functions are shown in Figures 4.7 and 4.8. For thousand runs, the average, the mean and the median were high in comparison to not using the inline implementation.

Template functions
Template functions can decrease the performance of software applications. According to Jacob B. Matthews, from the university of Chicago, [87], the use of templates decrease the efficiency of C++ compilers.

Virtual functions
The use of virtual functions has a cost for dispatching method calls. The work of [32] shows the overhead related with dispatching in C++ code.

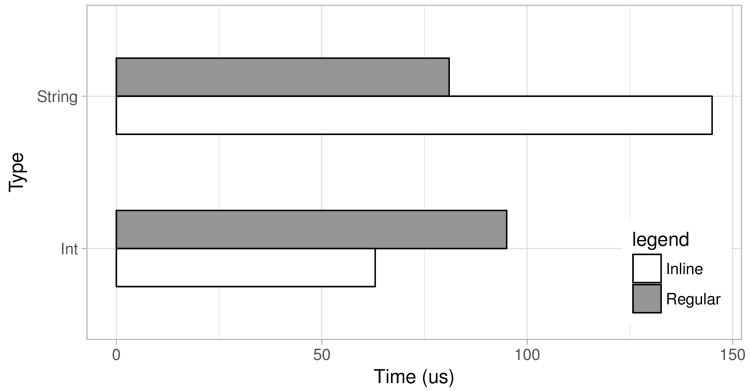

Figure 4.7 Counter Metrics for Inline and Regular function calls

4.8 Illustrative Example

Open Close To illustrate the approach used, we developed a small code which repeatedly opens a file. The code was instrumented with -finstrument functions and this gave the possibility to run the code using Lttng-UST. When this code is executed several times, some executions have a different behaviour, which we call outliers. Using our approach, we first recorded the executing metrics of the program, then the automated clustering was performed, and finally we compared the groups: slow executions and fast executions. We deduced that the problem was related with the wait-cpu time metric on the slow executions.

Figure 4.9 demonstrates the grouping mechanism, which segregated the executions into two groups, the association rule related all the slow executions with wait-cpu, group 2, and the fast executions with wait-cpu, group 1. The association showed that the wait-cpu was the cause for this difference between the groups.

Executions All experiments were conducted on a computer with a quad-core Intel® CoreTM i7-3770 CPU running at 3.4 GHz, 16 GB of DDR3 memory and a 7200 RPM hard drive. The Linux kernel version was 3.13.0-49 while the LTTng version was 2.6.0.

4.9 Case Studies

We used our approach on the following use cases:

4.9.1 Regression Comparison

A software company is releasing a new version of its software with several new features. The performance regression tests did not find problems and the unit tests were accepted. Some

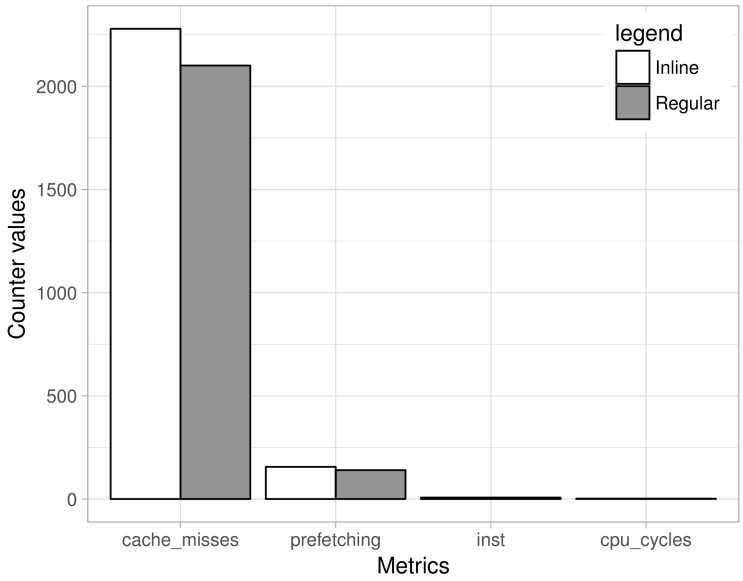

Figure 4.8 Distribution of Inline vs Regular functions using String and Integer as return types

weeks after the release, the users started to complain about performance issues in one of the features.

Approach: Our approach was to run the software several times with the previous and the current version of the software. Then, the data was classified using our approach into slow and fast executions.

Results: We were able to find that the addition of function inlining increased the number of cache misses and therefore increased the elapsed time of this application. In the source code, the inline functions could be seen as a difference, although the developers assumed that this could only improve the performance.

```
//header file
#ifndef EXAMPLE_H
#define EXAMPLE_H
/*
 *  Function included in multiple source
 *  files must be inline
 */
inline int sum(int a, int b) {
    return a + b;
}
inline void tolower(char* str) {
    for(int i = 0; str[i]; i++)
      str[i] = tolower(str[i]);
```

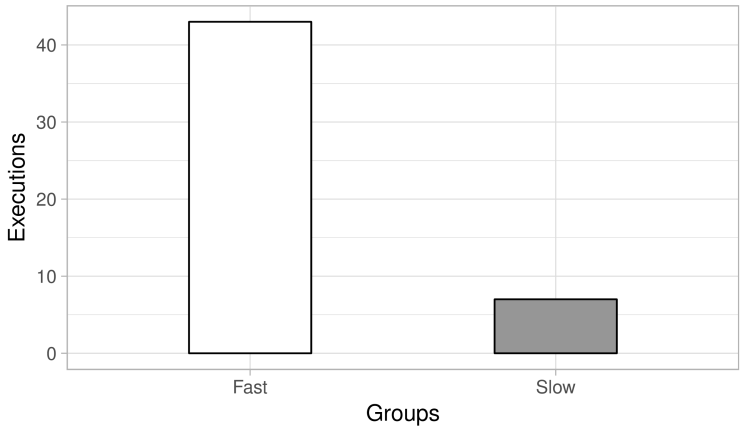

Figure 4.9 Comparing executions of the Open program

```
    }
    #endif
```

Code 4.1 Example of Code for inline

Using a comparative approach of the groups, we could see the differences in the metrics between each group. Comparing the groups of inline functions and the groups without inlining, we discovered that the inline groups comparatively had significantly more cache misses. We could conclude that cache misses were the essence of the differences, as shown in Table 4.2.

Table 4.2 Grouping results relating the cache misses with the slow executions groups

Executions	
Fast group	Slow group
mean of 2500 cache misses	mean of 6500 cache misses

4.9.2 Page Faults Interference

A software company deployed several new packages in its software stack. Those changes included a function implementation, which writes several times into a buffer. This problem usually occurs after several consecutive executions. Performance tools were able to discover a performance impact related to these new changes, but it was not possible to track specifically which commit caused the impact. The company would need to run specific Linux commands

to find the reason for this problem or reproduce several performance tests with each commit. This scenario was taken from [117], also related to a bug in [14].

Approach: Our approach was to instrument the code and to run the software several times with the previous and the current version of the software. Then, we classified the data using our approach in slow and fast executions, and compared the groups pair-wise. Although the auto-grouping gave more groups, i.e. reduction of ten to two groups of comparison, the aim was to specifically compare two groups of executions: fast and slow group.

Results: The tree generated had branches with fast and slow executions aggregated together. The nodes recorded the several performance metrics including the instructions, cache misses and page faults. We were able to find that extra page faults correlated with the decreased performance. The tool revealed the association, and we could conclude that the program triggered more page faults specifically after several executions. In the source code, a change in the buffer was an array that was implemented using a row major storage scheme and caused the addition of about 16 384 page faults. The company was not able to track this problem before, because of the prefetching algorithms influencing the executions. This technique is used in the microprocessor architecture to speedup the instructions.

The highlighted code shows the difference in the array coding that created the page faults difference. This code was taken from [117].

```
    //Considering 128 words/page
    int i, j;
    int data[128][128];
    for(j = 0; j < 128; j++)
        for(i = 0; i < 128; i++)
-           data[i][j] = 0;
+           data[j][i] = 0;
```

Code 4.2 Page fault interference - code changes

4.9.3 Cache Optimization in Server Application

A server application, using the well known PHP content-management framework Drupal, caches requested data to improve the access time to its content. However, we observed that after 100 requests, the request response time increased dramatically. Yet, no change was introduced in the software or hardware system, which is difficult to understand, as shown in Figure 4.10.

Drupal implements a caching mechanism which intends to improve the access performance.

In some executions, we noticed the influence of the compile time, which is an infrequent behavior of PHP.

Approach: First, we instrumented the PHP runtime and the Apache server, to be able to measure its behavior. Our approach was to execute several times a request for a server. While running it, we recorded the tracing data. Then, we executed our clustering analysis and classified the data into several groups, to study their behavior. Using this approach, it was possible to track infrequent issues in the execution. Indeed, if we have one or two groups with a totally different behavior, it is possible to compare their properties.

Results: From the collected data, we applied the auto-classification, which showed mainly two groups for comparison. The solution was able to display that, on the fast group, no time was spent on caching or PHP compilation time. However, in slow executions, there was a considerable amount of time spent on compilation time and caching, about 49%, as shown with the black bars in Figure 4.10. The approach was able to show the impact of disk access overhead related to the caching mechanism.
The problem was the naive implementation of the cache replacement strategy. When the cache is full, its whole content is simply flushed and must be built again for the most part. Therefore, every approximately 100 requests, the cache would fill and the server spent much more time than on the previous requests, as shown in Figure 4.10. The results are summarized in Figure 4.11.

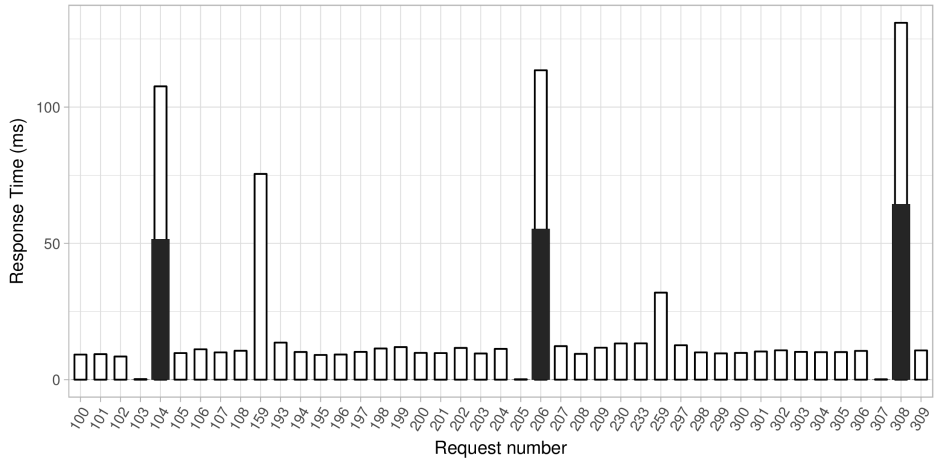

Figure 4.10 Overhead introduced by I/O for caching on each 100 of requests, The white bars are the request response time and the dark bars represent the PHP compilation time

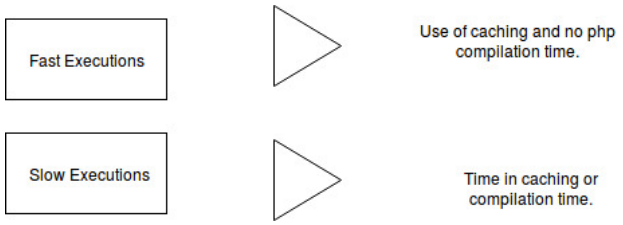

Figure 4.11 Difference on the groups

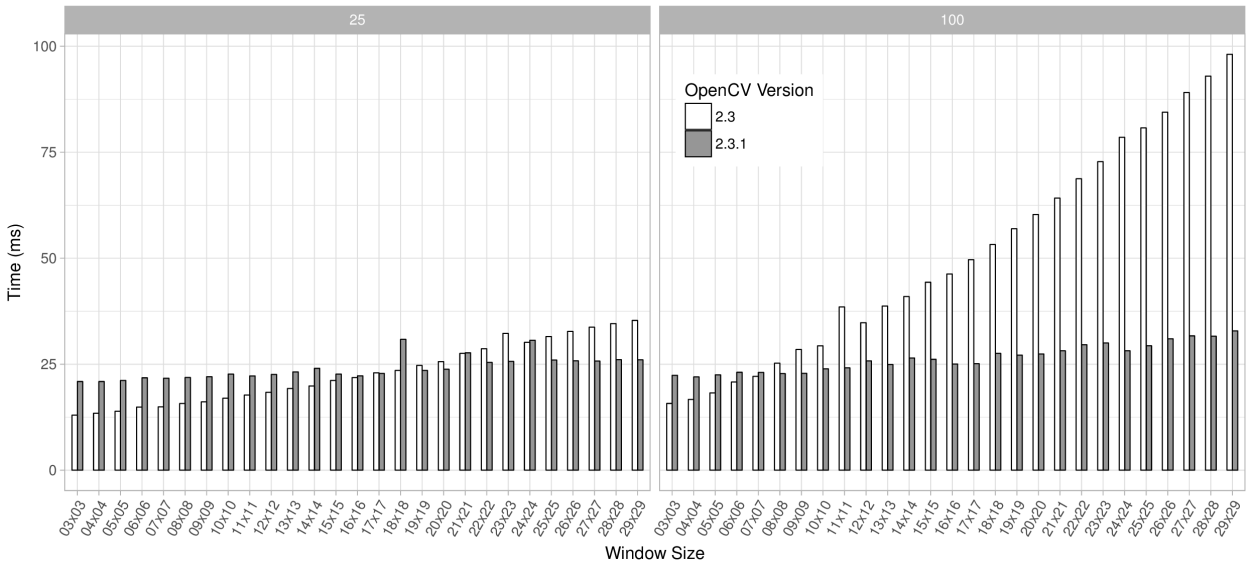

Figure 4.12 OpticalFlow Performance Regressions

4.9.4 OpenCV

The Open Source Computer Vision Library (OpenCV) is an open source computer vision and machine learning software library. This library is used in different problems in computer vision such as image tracking. In this section, we will benchmark the Optical Flow and HoughLines algorithms, which are the most evolving features in the recent years.

Optical Flow

The Optical Flow, implemented as the Lucas-Kanade algorithm, is an example of method in OpenCV and aims to correlate the apparent motion of objects between two consecutive frames. From some examples in the book [12], this method can be tested with two images, showing how they differ. Regressions can be caused by a series of changes in the code. Doing tests, we found a relevant regression in the Optical Flow function.

Figure 4.12 explores the regressions in this function, showing the performance of both versions according to the window size. It is interesting to highlight the fact that the performance of the newer version is better than the previous one until a certain point, where the previous version overpasses the newer one.

Approach: Our approach was to run the software several times, recording performance metrics using Linux Perf Events. The elapsed time to track the performance is also used in the classification. The runs were related with several versions of OpenCV until a regression was found between versions 2.3.0 and 2.3.1; the latest version was around 5 times slower than others.

Results: we used the approach described above, where the nodes of the tree have the frames of the OpenCV function calls. After carefully analyzing the data, the nodes for the newer version, considering all the other metrics, had more instructions. Thus, there was an association between the longer duration with more instructions.
Later, we verified the existence of a conditional statement differing from one version to another, which makes the slower version execute more instructions. This behaviour was originally reported in [103]. The total number of commits on those two versions were about 250 commits. Using this technique, we were able to reduce the scope of the significant difference to a few lines of code, and unit tests can be added easily to trigger specifically this cause, after knowing that it was a cache misses problem.

Table 4.3 Correlation among metrics

	inst.	cache misses	page faults	sched switches	prefetching
instructions	1	X	X	X	X
cache misses	**0.957**	1	X	X	X
page faults	**0.999**	**0.956**	1	X	X
sched switches	0.022	0.004	0.0004	1	X
prefetching	**0.996**	**0.969**	**0.995**	0.014	1

In the Table 4.3 the Pearson correlation (R) of the variables is presented, which shows that many metrics are directly proportional, i.e. R bigger than 0.75. This mean that the development of models using those metrics, for example linear or multilinear models, is not

adequate and consequently the comparison of groups, in a pair-wise approach, can be applied. Figure 4.13 shows the tested image of the optical flow from two images.

Figure 4.13 Optical Flow Example

HoughLines

HoughLines is a line detection mechanism implemented in OpenCV, to find edges in images. This can be used to find edges in roads and it is implemented using an algorithm called Canny86. While comparing different versions of OpenCV, we found that version 3.1 takes more time than version 3.0 in a case of Software Regression [37]. The source code difference can be found here [102].

Approach: Our approach was to run the performance unit tests several times, recording performance metrics using Linux Perf Events in a large file. The elapsed time to track the performance is also used in the classification. The runs were related with several versions of OpenCV, until we found the regression between versions 3.0 and 3.1, where the newer version was slower than the previous one.

Results: The results correlate the longer duration with cache misses. By analyzing the code, we were able to find a difference in file hough.cpp, related with a different size of the array accessed. This process reduced considerably the search for the causation, helping in the development of a solution.

Figure 4.14, shows the grouping results, relating the cache misses with the slow executions groups. This figure represents the fact that all the groups with low cache misses were fast, therefore the fast and low cache misses circles override. Consequently, the circles with high cache misses and slow executions also override.

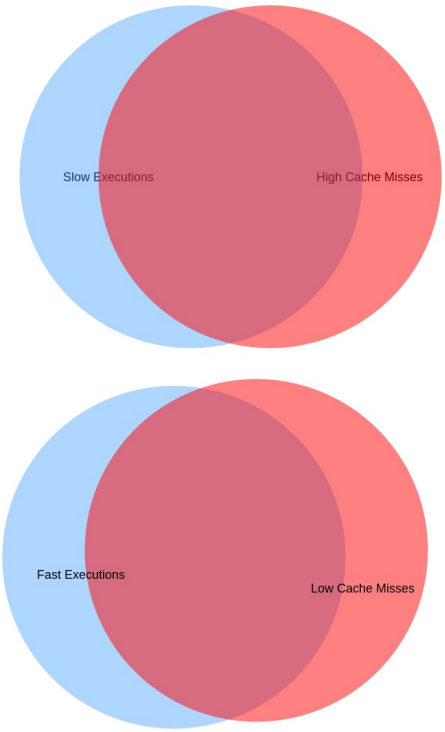

Figure 4.14 Case study Regression - showing significantly differences in the groups of runs in terms of metrics.

4.10 Discussion

This section presents a discussion related with the research questions presented in the introduction, as well as with the solution presented. The results of the use cases clarify the research questions and bring new insights on the performance.

- RQ 1: How can we build an efficient and flexible model for performance comparison?
  Using the enhanced calling context tree, it is possible to aggregate performance metrics and compare executions efficiently. The CCT can be built with or without source code modification. The possibility to build it without source code brings the opportunity to analyze the dynamic structure, and an accurate behaviour can be used.

- RQ 2: Is it possible to automate the performance analysis?
  Our work demonstrated the possibility to use non-supervised machine learning methods to compare and find performance problems, specifically using a comparative heuristic technique. The approach uses a comparative clustering mechanism, which can efficiently separate the data, for a realistic comparison.

- RQ 3: How accurate were the obtained results?
  The methodology used was able to find the association between groups of metrics, and consequently associate the cause of elapsed time variations. However, this approach can have type I or type II errors - false positives and false negatives.

4.11 Threats to Validity

This section discusses the threats to the validity of our study.
Time analysis Our approach is based on comparing several clusters, to take the best grouping according to the SSE, which requires a comparatively long period of time. Our study aims to use an automated non-supervised clustering method, thus the analysis time is a minor factor if it reduces the human analysis time. The analysis required just a few minutes, between building the ECCT tree and classifying the metrics. Another highlight is that the analysis is made offline, so the time will not influence the performance of the software and does not require stopping the software development.

Quantity of groups Since the automated heuristic used can generate a high number of groups, this can increase the difficulty of the evaluation.

Dimensional Analysis Continuing previous work, the technique was applied just in two dimensions each time (i.e. time and a metric n), which is a severe restriction for data mining techniques. The next step would be to use several dimensions at the same time.

4.12 Future Work

As future work, we plan to expand our investigation by taking several dimensions in consideration, then by using non-linear models to track regression problems in different software versions[53]. This can be used as an automated test to find software regressions. An example of possible models are feed-forward networks, also called deep learning networks. The models need to be able to characterize specifically non-linear dependencies and be able to be used without labeled data.

Although we partially implemented this approach, another possibility would be to combine other techniques such as the Apriori algorithm, [3], which can determine association rules among the metrics recorded in the enhanced CCT for each cluster. Finally, tracking performance issues before the release of new software is an interesting path to be followed. Thus,

an automated mechanism to find them before the release of a new software version is very promising. A possibility could be to develop a mechanism to be executed as a regression test suite, from the machine learning models described above, before each release.

4.13 Conclusion

In conclusion, this research developed a solution that showed the possibility of using clustering mechanisms, without human intervention, to find causes of performance problems. As a contribution for developers, this paper introduces the visualization tool for the Calling Context Tree, with Flame Graphs and Auto Cluster mechanisms.
The clustering data was built through a bottom-up analysis on collected stack traces, from recorded trace data on ECCTs. This data structure was implemented in the CCT View, inside the Trace Compass framework, and provides several run-time properties of the studied software.

The solution presents an automated algorithm based on heuristic comparison. This approach is the first one to target performance analysis on trace executions and call graph data. The solution also proposed a flexible approach which can be used combined with different means to collect the data, such as LTTng and Perf counter to collect the data that will be analyzed, as was done in some use cases.

The grouping approach used was used without supervision. This is an advantage over techniques that requires a training phase. For example, Support Vector Machines require data pre-classification.

This paper compared the different techniques that can be used for this kind of operation. Some of the techniques compared required another level of data processing to achieve non-supervised clustering, for example the SVM technique, yet they could be included in this comparison of auto clustering techniques.

Our work was able to find causes of performance issues without human intervention, and can be applied to other cases to find other problems. The implementation as part of the Trace Compass framework aims to apply this approach for more cases and large scale software analysis.

4.14 Acknowledgement

The author would like to thank the Council of Research of Canada and the Dorsal lab. Special thanks to Naser Ezzati and Genevieve Bastien, for comments and suggestions throughout the development of this solution.

CHAPTER 5 GENERAL DISCUSSION

The demand for performance is increasing, particularly for the software part where lies a significant portion of the developer controlled algorithmic complexity. Consequently, the new proposed performance analysis technique can be used for this purpose. In this context, this research aims to demonstrate a new perspective on system analysis, in the automated identification of performance root causes, and proposes a tool that should be especially useful for performance improvement.
The proposed solution is an autonomous technique, which combines tracing and clustering, to find causes of performance issues on real programs. This research can still be relevant even with software updates and obsolescence, since it relies on a methodology rather than a tool. Moreover, the heuristic utilization, elbow method, is able to fulfill this requirement, since it can be applied with different classification methods and arbitrary dimensions.
Considering the present situation, with tracing and performance counters, this work could be applied as is, in different environments, and could even be integrated into a testing framework such as PARCS. Although some adaptations will be needed in the data collection part, the main part of this research can be used in other domains as well.
The development of better PADBI tools, Performance Anomaly Detection and Bottleneck Identification, is eventually leading towards automated mechanisms and methods. The data mining methods presented in this work aim to partially fulfill this gap.

5.1 Methods comparison

5.1.1 Clustering methods

Since several methods were applied to the task, it is possible to compare them in several ways, which are summarized below. As explained in the article, the SVM model delimits an hyperplane to divide the data and, as a consequence, is able to segregate the slow and fast executions. In some scenarios, especially considering the influence of outliers, the classification can be mislead and produce strange segregated data results.

The second method, percentage classification, is an unsupervised mechanism and leads to segregation of data, even if they are homogeneously distributed among the dataset. The percentage classification is interesting to use in scenarios similar to k-means, without the unpredictability of the original method.

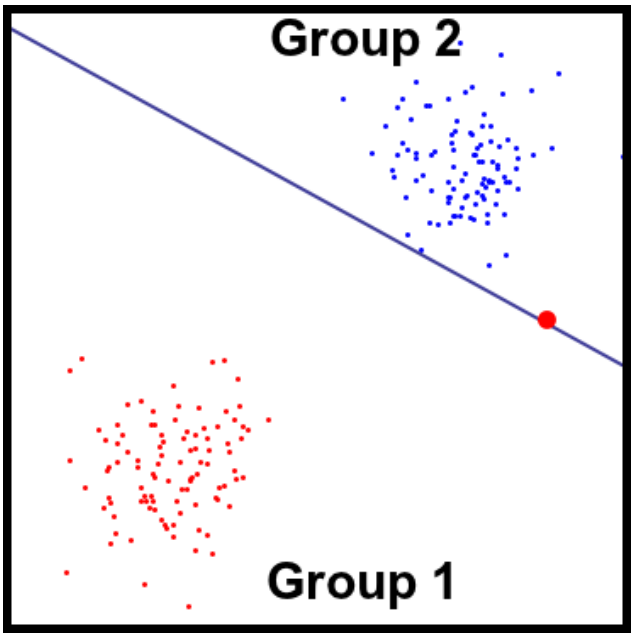

Figure 5.1 Support Vector Scenarios

This method is interesting when the standard deviations of the data are similar, so the result will be a reduced number of groups. However, it may lead to a single group or to an unduly large number of groups when the average distance is too low or high as compared to a certain threshold. This method ties the solution of a specific problem to the quantification of the groups in a distribution, scenario 1 in Figure 5.1.

Finally, the k-means algorithm is efficient but requires to specify the number of groups to be used in the classification process, which clusters the data using a centroid. Still, k-means does not carry the exact same result in each run. The k-means algorithm is especially interesting in scenarios such as with two or three main clusters. However, it is less interesting in scenarios where the data is homogeneously distributed and when there is just one cluster.

The scenarios described in Figure 5.3, [109], are clear limitations of the k-means algorithm. The first scenario is when the data is not suitable for this technique. In our experiments, neither one of the scenarios were found in the use cases.

The methods described in this work summarize a series of methods to measure differences among groups of execution. The data mining methods were SVM and k-means. On the other hand, the statistical method used in the solution can provide a relative comparison of the performance among the groups, to specifically identify the metrics related to the root cause for issues.

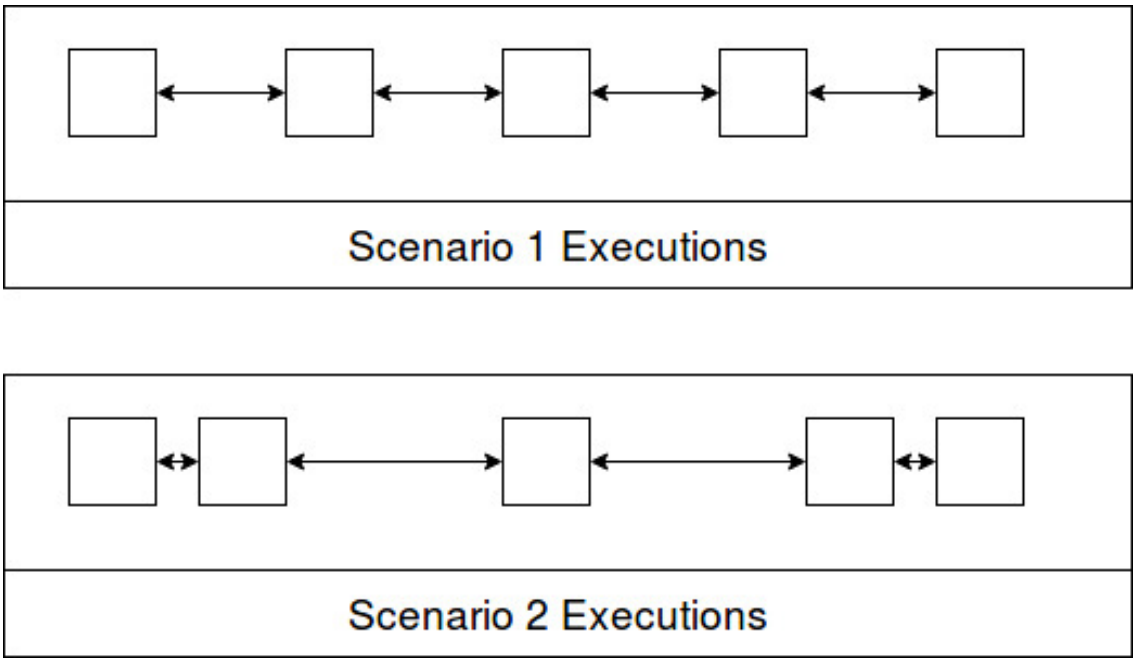

Figure 5.2 Percentage Classification Scenarios

The statistical tools can also be used to detect regressions, as the collaboration [1] showed, using confidence Interval to find regressions among Chromium releases. As a final remark, the comparison is whether the groups are well defined and whether the anomalies are significant and not just a new pattern in the data, as listed in [16]. In this matter, each grouping technique, such as the percentage grouping, has its own border definition.

5.1.2 Classification strategies

From the point of view of the different strategies used to segregate the executions, before the creating of groups in the pre-processing phase, the percentage strategies can split the data depending on the mean separation of the executions. This is shown in Figure 5.2: in the scenario 1, the executions are equally spaced, they will be in the same group. However, in scenario 2, the executions will not in the same group.

5.2 Results discussion

The results were able to indicate specific metrics that might be used to improve software performance. The solution can be used in two ways: as a detection tool or as an analysis tool. In other words, in some cases the solution is able specifically to demonstrate where corrections should be done, while in other cases the solution will give a guideline to a specific

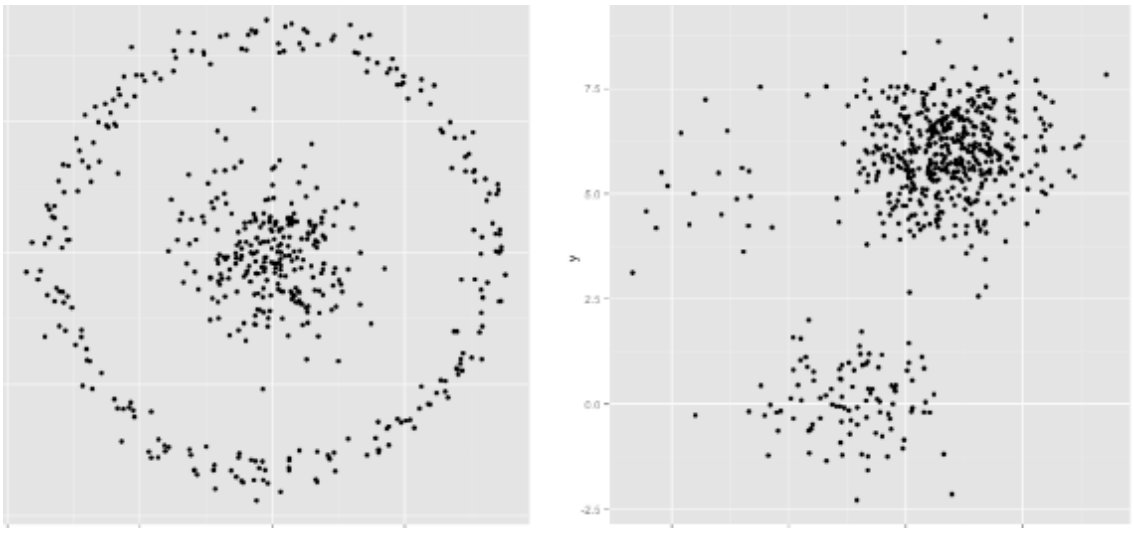

Figure 5.3 K-means Scenarios [source [109]]

metric related to the issue, i.e. a metric guidance that can be used by the developer to find the relevant source code location.
In the OpenCV example, it was possible to reduce the number of commits using the result in a specific function. However, it was necessary to consider the code of the application to do a complete analysis.
In other cases, the longer runs were related to specific metrics, e.g. cache misses, and the groups comparison was able to identify this information. The versions could then be compared considering this information, so the regressions test can be properly done.

5.3 Metrics collision

During the experiments, we found that a larger overhead was added when using several performance metrics in tracing at the same time, i.e. there is a direct relation between more metrics and the overhead. Therefore, instead of recording all the metrics at the same time, it is possible to use an experimental setup to infer the influence of different metrics in a application.
This method is not presented in the paper, since we were able to trace the applications with all the metrics enabled. Using the cross validation setup, the results can be summarized in the tables below:

Using a simplification of the Apriori algorithm, considering the results in the following Table 5.2, knowing that the impact of BC is 40 percent and the impact of AB is 10 percent, the deduced impact of C must be no less than 20 percent.

Table 5.1 Metric Association

Metric A	Metric B	Metric C
enabled	-	enabled
-	enabled	enabled
enabled	enabled	-

Table 5.2 Group Impact

Groups	Impact
AB	10 percent impact
AC	40 percent impact
BC	30 percent impact

It is possible therefore to construct an association rule to simplify this process and avoid unnecessary tracing overhead.

5.4 Auto clustering discussion

Considering the models shown above, we chose to develop a non-supervised method, a.k.a auto clustering. The possibility to use an automated approach is more interesting for us, mainly because we aim to avoid using data to train a model.
Therefore, we implemented a version of comparative k-means using the SSE (sum of square errors) variability information, plus a heuristic evaluation. This technique can be used for an arbitrary dimension, since the amount of difference, SSE (sum of squared error), can be calculated for those cases. Some limitations apply to this approach and it is possible to use the Gaussian Mixture Model to overcome these, as elaborated in [127].
Since the SSE drops considerably between one and two groups, we always need to consider more than two groups. Otherwise, the chosen gap will be two groups. From the perspective of the Elbow method, also called silhouette method, this is in fact just a criteria used to find the best k, such as the Calinski criterion as described in [13]. This approach, Calinski-Harabasz, is used in [114] to specifically determine the number of clusters.
The approach used can also be compared with the x-means clustering technique, where the best results are kept to be compared. A more complete approach could be obtained using the statistical gap calculation, which computes the dispersion within a cluster, as described

in [128]. The next step would be to use an agglomerative clustering technique, which is a bottom up technique.

5.5 Performance Counters

A first important consideration in this research is the selection of performance counters. More metrics may help uncover problems but they cumulatively add to the tracing overhead. The cross-validation scheme may help in the selection, depending on the number of available metrics. The process of grouping will increase its complexity, when hundreds of metrics are available, as expressed in [114].
Besides, with hundreds of metrics, the pair-wise analysis used in the groups comparison could be a limitation, consequently another control chart could be used.

5.6 Data structure evaluation

We used two dynamic structures in the solution, a calling context tree with metrics, ECCT, but also a dynamic call tree with metrics, EDT. The main difference resides in the summarized purpose of the ECCT, which was used in previous work. Although the EDT has a much larger size, it avoids the addition of a pre-processing phase in the comparison of the internal nodes. Thus, when comparing small portions of a software application, where an entire tree is unnecessary, the use of EDT can be more efficient.
Those two dynamic structures, as other trees, can be used in comparison analysis in two specific ways: the topological organization of the structure and the weights of the nodes.
The first approach can be used to compare differences among software executions, specifically targeting the profiling approaches. This strategy can be used in terms of trace mining, in kernel space, if a pattern matching approach is used to automatically delimit the entries and exits of the nodes.
The second approach was used in this research and consists in comparing the weights of nodes from similar trees. In this approach, the weights are the hardware and software performance metrics, e.g. page-faults and cache-misses.

5.7 Usage

With this tool, there are various situations where standard profiling is not suitable and tracing requires substantial analysis, for example with large traces. An unequivocal example of real utilization of this tool is the regressions tests, as shown in the use cases of OpenCV

and cache-misses. Although the proposed solution can be used to detect regressions, it is suitable as a diagnosis tool as well, since it can indicate the root cause.
The solution presented in this research could be integrated to PARCS, or even in similar frameworks, where the performance metrics are compared in each version, to see the patterns, since the mining technique has a complementary approach that compares the weights of the nodes, instead of the topological structure.
Although not explored, it is possible to use this work for developing enhanced trees, ECCTs or EDTs, for the kernel side of the system, and applying the same comparison techniques explored here.

5.8 Visualization Tools

The Red Grey Green Differential graphs, RGG, suggested in the work, can easily highlight the differences between similar executions, which would have the same color although they had different relative performance.
This different implementation, in comparison to the standard Differential Flame Graph, will reduce the analysis time by avoiding the verification of the equal time use cases.
We conclude this research through an evaluative summary of the contribution, by considering several aspects of the proposed solution. First, we present a summary of the work, then the objectives fulfillment, followed by the limitations of our solution and recommendations for future improvements.

CHAPTER 6 CONCLUSION

We conclude this demonstration of the work by summarizing our contributions. Then, we present the limitations of our solution and recommend future improvements.

6.1 Summary of the work

The current tools are limited by the quantity of data that can be analyzed by users. Therefore, several data mining techniques can be applied to find associations and correlations among executions. This research aims to expand the current models by introducing an automated comparison mechanism.
The contribution consists in an automated mechanism, using a heuristic evaluation to automate grouping techniques. The heuristic evaluation can be built upon several already used algorithms, such as k-means. Different comparison strategies, such as the percentage comparison may highlight different behaviours of the system in a complementary way.
The proposed solution compares the groups of executions with different performance and evaluates their inner structure, using or not the source code as reference. This work is composed of data mining algorithms, heuristic validation, the association rule and statistical methods.

6.2 Objective attainment

This work aimed to demonstrate the possibility to use automatic methods to reduce considerably the analysis time for performance comparison. The proposed solution, the auto grouping mechanism presented in the article, enabled an automatic comparison of groups of executions and a comparison of their behavior. Finally, the solution could be applied in several case studies, demonstrating the effectiveness of the approach in order to find performance issues with a group comparison strategy. This allowed to conclude that the challenge overall is achievable.

6.3 Contributions

The scientific contributions for this work were summarized in the papers presented.
(1) We used a heuristic comparison to automate the current cluster and grouping mecha-

nism, without unduly increasing the complexity of those algorithms. Using this strategy, we were able to propose an automatic way to compare several executions, without necessarily selecting the number of groups that would be compared.
(2) Comparison of several clustering techniques, analyzing their advantages and disadvantages. This may be used in the future to improve those methods.
(3) We developed the RGG Differential flame graph, which can show differences on the executions in three colors, reducing the ambiguity of the comparison of the original DFG. This is a new view that allows comparing two groups of executions.
(4) In co-authorship, we applied Confidence Intervals using medians, instead of means, to compare several releases of the Chromium software. This technique was able to overpass the current CI implementations. (5) Bing data mining to tracing in order to evaluate performance.

6.4 Limitations of the solution

The auto grouping technique has two main limitations: the causation and the optimal number of groups. First, an association with the group analysis does not necessarily imply causation, since correlation does not mean causation. A deeper evaluation would be necessary to study the causality and its association with the performance counters, in terms of real percentage of representation and causation.
The second limitation in the auto-grouping, as well as group analysis in general, is the optimal number of groups, which cannot necessarily facilitate the human analysis of the system. Sometimes, the number of groups is particularly large and, although the number is optimal for some criteria, it is not necessarily the best number in terms of performance analysis.

6.5 Future Work

In the future, it is possible to develop a fully automatic analysis framework, that includes trace correlation, comparison and statistical analysis. Although not demonstrated in the use cases, this work can be used to compare fuzzy data, i.e. more than two groups of comparison automatically. However a deep analysis with each metric could be performed to verify and measure specifically the impact of each performance variation in the runs in real workloads. A promising area is to use a neural network or other techniques to correlate the userspace traces with kernel space traces. This could highlight interesting aspects of performance evaluation. Furthermore, this technique does not require the use of labeled data. The

solution was implemented focusing on userspace tracing, and can only capture the stack for ELF binaries. Therefore, it could be extended to address other needs such as JIT compilation.

REFERENCES

[1] B. A. Abderrahmane Benbachir, Isnaldo Francisco De Melo jr and M. Dagenais. Automated performance deviation detection across google chromium releases. In *QRS 2017*, July 2017.

[2] A. Adamoli and M. Hauswirth. Trevis: A context tree visualization & analysis framework and its use for classifying performance failure reportsn. In *Proceedings of the 5th International Symposium on Software Visualization*, SOFTVIS '10, pages 73–82, New York, NY, USA, 2010. ACM.

[3] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. proceedings of the 20th international conference on very large data bases, vldb, pages 487-499, santiago, chile, september 1994., May 2017.

[4] R. R. Alexandre Bergel, Felipe Banados and W. Binder. Execution profiling blueprints. software: Practice and experience. *SOFTWARE—PRACTICE AND EXPERIENCE 00:1–29*, page 231–244, 2010.

[5] D. C. L. W. N. W. BA Wichmann, AA. Canning and D. Marsh. Industrial perspective on static analysis. *Software Engineering Journal*, Mar. 1995.

[6] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using magpie for request extraction and workload modelling. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 18–18, Berkeley, CA, USA, 2004. USENIX Association.

[7] P. Beaucamps, I. Gnaedig, and J.-Y. Marion. Behavior abstraction in malware analysis. In H. Barringer, Y. Falcone, B. Finkbeiner, K. Havelund, I. Lee, G. Pace, G. Roşu, O. Sokolsky, and N. Tillmann, editors, *Runtime Verification: First International Conference, RV 2010, St. Julians, Malta, November 1-4, 2010. Proceedings*, pages 168–182, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[8] T. Bertauld and M. R. Dagenais. Low-level trace correlation on heterogeneous embedded systems. *EURASIP Journal on Embedded Systems*, 2017(1):18, 2017.

[9] C. Bezemer, E. Milon, A. Zaidman, and J. Pouwelse. Detecting and analyzing i/o performance regressions. *J. Softw. Evol. Process*, 26(12):1193–1212, Dec. 2014.

[10] C. Bielza and P. Larrañaga. Discrete bayesian network classifiers: A survey. *ACM Comput. Surv.*, 47(1):5:1–5:43, July 2014.

[11] S. M. Blackburn, A. Diwan, M. Hauswirth, P. F. Sweeney, J. N. Amaral, T. Brecht, L. Bulej, C. Click, L. Eeckhout, S. Fischmeister, D. Frampton, L. J. Hendren, M. Hind, A. L. Hosking, R. E. Jones, T. Kalibera, N. Keynes, N. Nystrom, and A. Zeller. The truth, the whole truth, and nothing but the truth: A pragmatic guide to assessing empirical evaluations. *ACM Trans. Program. Lang. Syst.*, 38(4):15:1–15:20, Oct. 2016.

[12] G. Bradski and A. Kaehler. *Learning OpenCV 2*. O'Reilly Media, 2007.

[13] T. Caliński and J. Harabasz. A dendrite method for cluster analysis. *Communications in Statistics-Simulation and Computation*, 3(1):1–27, 1974.

[14] M. D. N. Casablanca. Memory leak issue with casablanca library, 2013. https://social.msdn.microsoft.com/Forums/en-US/b9d5b83a-2465-4db1-b292-23b5ea6a6ef2/memory-leak-issue-with-casablanca-library?forum=casablanca.

[15] B. S. Center. Paraver. `https://tools.bsc.es/paraver`, 2017. Accessed: 2017-04-30.

[16] V. Chandola, A. Banerjee, and V. Kumar. Anomaly detection: A survey. *ACM Comput. Surv.*, 41(3):15:1–15:58, July 2009.

[17] M. Chen, A. X. Zheng, J. Lloyd, M. I. Jordan, and E. Brewer. Failure diagnosis using decision trees. In *Proceeding ICAC '04 Proceedings of the First International Conference on Autonomic Computing*, pages 36–43, 2004.

[18] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, DSN '02, pages 595–604, Washington, DC, USA, 2002. IEEE Computer Society.

[19] G. Chromium. Tracing ecosystem explainer. `https://docs.google.com/document/d/1QADiFe0ss7Ydq-LUNOPpIf6z4KXGuWs_ygxiJxoMZKo/edit`, 2015. Accessed: 2017-04-30.

[20] D. L. P. D. S. Cito, J.; Suljoti. Identifying root causes of web performance degradation using changepoint analysis. In *14th International Conference on Web Engineering*. ICWE, 2014.

[21] I. Cohen, F. G. Cozman, N. Sebe, M. C. Cirelo, and T. S. Huang. Semisupervised learning of classifiers: Theory, algorithms, and their application to human-computer interaction. *IEEE Trans. Pattern Anal. Mach. Intell.*, 26(12):1553–1567, Dec. 2004.

[22] E. T. Compass. Trace compass. `http://tracecompass.org/"`, May 2017. [Online; accessed 05-May-2017].

[23] K. Das. *Detecting patterns of anomalies*. PhD thesis, Carnegie Mellon University, Mar. 2009.

[24] F. Deng, N. DiGiuseppe, and J. A. Jones. Constellation visualization: Augmenting program dependence with dynamic information. In *2011 6th International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT)*, pages 1–8, Sept 2011.

[25] M. Desnoyers. *Low-impact Operating System Tracing*. PhD thesis, Ecole Polytechnique de Montreal, 2009.

[26] M. Desnoyers and M. R. Dagenais. The lttng tracer: A low impact performance and behavior monitor for gnu/linux, May 2006.

[27] M. Desnoyers and M. R. Dagenais. The lttng tracer: A low impact performance and behavior monitor for gnu/linux, 2006.

[28] F. Doray. Analyse de variations de performance par comparaison de traces d'exécution. Master's thesis, École Polytechnique de Montréal, Jul 2015.

[29] F. Doray. Tigerbeetle. `https://github.com/fdoray/tigerbeetle`, 2015. [Online; accessed 05-May-2017].

[30] F. Doray. Trace compare. `https://github.com/fdoray/tracecompare`, 2015. [Online; accessed 05-May-2017].

[31] F. Doray and M. Dagenais. Diagnosing performance variations by comparing multi-level execution traces. *IEEE Transactions on Parallel and Distributed Systems*, 28(2):462–474, Feb 2017.

[32] K. Driesen and U. Hölzle. The direct cost of virtual function calls in c++, 1996. Presented at OOPSLA 96 - 10/96, San Jose, CA.

[33] Dtrace.org. Dtrace. `http://dtrace.org/blogs/about/`, 2017. Accessed: 2017-04-30.

[34] S. T. Eckmann, G. Vigna, and R. A. Kemmerer. Statl: An attack language for state-based intrusion detection. *J. Comput. Secur.*, 10(1-2):71–103, July 2002.

[35] A. Edwards, H. Vo, A. Srivastava, and A. Srivastava. Vulcan binary transformation in a distributed environment. Technical report, 2001.

[36] Efficios. Doc lttng. `http://lttng.org/docs/v2.9/#doc-tracing-session-mode`, 2017. Accessed: 2017-04-30.

[37] S. Emami. Time tests. `http://shervinemami.info/timingTests.html`, 2012. [Online; accessed 05-January-2017].

[38] Y. Endo and M. Seltzer. Improving interactive performance using tipme. *SIGMETRICS Perform. Eval. Rev.*, 28(1):240–251, June 2000.

[39] N. Ezzati-Jivan and M. Dagenais. An efficient analysis approach for multi-core system tracing data. In *Proceedings of the 16th IASTED International Conference on Software Engineering and Applications (SEA 2012)*, 2012.

[40] N. Ezzati-Jivan and M. R. Dagenais. A stateful approach to generate synthetic events from kernel traces. *Adv. Soft. Eng.*, 2012:6:6–6:6, Jan. 2012.

[41] N. Ezzati-Jivan and M. R. Dagenais. A framework to compute statistics of system parameters from very large trace files. *SIGOPS Oper. Syst. Rev.*, 47(1):43–54, Jan. 2013.

[42] A. Field. *Discovering statistics using SPSS*. SAGE Publications, 2013.

[43] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica. X-trace: A pervasive network tracing framework. In *Proceedings of the 4th USENIX Conference on Networked Systems Design & Implementation*, NSDI'07, pages 20–20, Berkeley, CA, USA, 2007. USENIX Association.

[44] P.-M. Fournier and M. R. Dagenais. Analyzing blocking to debug performance problems on multi-core systems. *SIGOPS Oper. Syst. Rev.*, 44(2):77–87, Apr. 2010.

[45] M. Fredrikson, M. Christodorescu, and S. Jha. *Dynamic Behavior Matching: A Complexity Analysis and New Approximation Algorithms*, pages 252–267. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[46] N. Friedman, D. Geiger, and M. Goldszmidt. Bayesian network classifiers, Jan. 2009.

[47] T. B. G. Ammons and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling, in proceedings of the acm sigplan 1997 conference on programming language design and implementation, ser. pldi '97. new york, ny: Acm, 1997, pp. 85–96., Jul 2015.

[48] J. Gao, G. Jiang, H. Chen, and J. Han. Modeling probabilistic measurement correlations for problem determination in large-scale distributed systems. In *2009 29th IEEE International Conference on Distributed Computing Systems*, pages 623–630, June 2009.

[49] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous java performance evaluation. *SIGPLAN Not.*, 42(10):57–76, Oct. 2007.

[50] F. Giraldeau. *ANALYSE DE PERFORMANCE DE SYSTÈMES DISTRIBUÉS ET HÉTÉROGÈNES À L'AIDE DE TRAÇAGE NOYAU*. PhD thesis, École Polytechnique de Montréal, Nov. 2015.

[51] F. Giraldeau, J. Desfossez, D. Goulet, M. Dagenais, and M. Desnoyers. Recovering system metrics from kernel trace. *Linux Symposium*, 190:109–115, 2011.

[52] F. Giraldeau, J. Desfossez, D. Goulet, M. Desnoyers, and M. R. Dagenais. Recovering system metrics from kernel trace. In *Linux Symposium 2011*, June 2011.

[53] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[54] Google. Chromium project. `https://www.chromium.org/`, 2013. Accessed: 2017-04-30.

[55] S. Goswami. An introduction to kprobes. `https://lwn.net/Articles/132196/`, 2005. [Accessed 05-May-2017].

[56] S. L. Graham, P. B. Kessler, and M. K. McKusick. gprof: a call graph execution profiler, 1982.

[57] B. Gregg. *Systems performance : enterprise and the cloud*. Prentice Hall, 2007.

[58] B. Gregg. Differential flame graph. `http://www.brendangregg.com/blog/2014-11-09/differential-flame-graphs.html`, 2014. [Online; accessed 05-May-2017].

[59] B. Gregg. ebpf: One small step. `http://www.brendangregg.com/blog/2015-05-15/ebpf-one-small-step.html`, 2015. [Online; accessed 05-May-2017].

[60] B. Gregg. ebpf: One small step. http://www.brendangregg.com/blog/2015-05-15/ebpf-one-small-step.html, 2015. Accessed: 2017-04-30.

[61] B. Gregg. Differential flame graphs. http://www.brendangregg.com/blog/2014-11-09/differential-flame-graphs.html, May 2017. [Online; accessed 05-May-2017].

[62] B. Gregg. Flame graphs. http://www.brendangregg.com/flamegraphs.html, 2017. [Online; accessed 05-May-2017].

[63] S. I. Group. Better code hub. https://bettercodehub.com/, 2017. Accessed: 2017-04-30.

[64] K. Guntheroth. *Optimized C++*. O'Reilly Media, 2016.

[65] V. Gupta, M. Singh, and V. K. Bhalla. Pattern matching algorithms for intrusion detection and prevention system: A comparative analysis. In *2014 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, pages 50–54, Sept 2014.

[66] A. Hamou-Lhadj and T. Lethbridge. Summarizing the content of large traces to facilitate the understanding of the behaviour of a software system. In *14th IEEE International Conference on Program Comprehension (ICPC'06)*, pages 181–190, 2006.

[67] O. Ibidunmoye, F. Hernández-Rodriguez, and E. Elmroth. Performance anomaly detection and bottleneck identification. *ACM Comput. Surv.*, 48(1):4:1–4:35, July 2015.

[68] M. Idris, A. Mehrabian, A. Hamou-Lhadj, and R. Khoury. Pattern-based trace correlation technique to compare software versions. In *Proceedings of the Third International Conference on Autonomous and Intelligent Systems*, AIS'12, pages 159–166, Berlin, Heidelberg, 2012. Springer-Verlag.

[69] R. H. Inc. ftrace - function tracer. https://www.kernel.org/doc/Documentation/trace/ftrace.txt, 2015. Accessed: 2017-04-30.

[70] P. Isensee. Inline performance, 2016. [Online; accessed 05-May-2017].

[71] D. Jackson and D. A. Ladd. Semantic diff: A tool for summarizing the effects of modifications. In *Proceedings of the International Conference on Software Maintenance*, ICSM '94, pages 243–252, Washington, DC, USA, 1994. IEEE Computer Society.

[72] A. K. Jain and R. C. Dubes. *Algorithms for Clustering Data*. Prentice Hall, Englewood Cliffs, New Jersey, 1988.

[73] H. Kang, X. Zhu, and J. L. Wong. Dapa: diagnosing application performance anomalies for virtualized infrastructures. In *Presented as part of the 2nd USENIX Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services. USENIX*, pages 8–8, 2012.

[74] Kernel.org. perf: Linux profiling with performance counters, May 2014.

[75] M. Kerrisk. Bpf. `http://man7.org/linux/man-pages/man2/bpf.2.html`, 2016. Accessed: 2017-04-30.

[76] C. H. Kim, J. Rhee, H. Zhang, N. Arora, X. G. Jiang, Zhang, and D. Xu. Introperf: Transparent context-sensitive multi-layer performance inference using system stack traces. In *The 2014 ACM International Conference on Measurement and Modeling of Computer Systems*, pages 235–247, 2014.

[77] N. Kim. Uftrace, 2017. "accessed in 15-March-2017".

[78] A. F. Klaib and H. Osborne. Rsma matching algorithm for searching biological sequences. In *Proceedings of the 6th International Conference on Innovations in Information Technology*, IIT'09, pages 190–194, Piscataway, NJ, USA, 2009. IEEE Press.

[79] M. F. M. K. G. M. J. M. C. L. Adhianto, S. Banerjee and N. R. Tallent. Hpctoolkit: tools for performance analysis of optimized parallel programs," concurrency and computation: Practice and experience, vol. 22, no. 6, pp. 685–701, apr.2010., Jul 2015.

[80] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. *SIGPLAN Not.*, 40(6):190–200, June 2005.

[81] J. Mace. End-to-End Tracing: Adoption and Use Cases. Survey, Brown University, 2017.

[82] J. Maebe, M. Ronsse, and K. D. Bosschere. Diota: Dynamic instrumentation, optimization and transformation of applications. In *IN PROC. 4TH WORKSHOP ON BINARY TRANSLATION (WBT'02)*, 2002.

[83] J. a. P. Magalhães and L. M. Silva. Root-cause analysis of performance anomalies in web-based applications. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, SAC '11, pages 209–216, New York, NY, USA, 2011. ACM.

[84] U. Manpage. Tracediff. http://manpages.ubuntu.com/manpages/precise/man1/tracediff.1.html, 2017. [Online; accessed 05-May-2017].

[85] R. S. Martin Bligh, Mathieu Desnoyers. Linux kernel debugging on google sized clusters. In *OLS (Ottawa Linux Symposium)*, page 29–40, 2007.

[86] G. Matni and M. Dagenais. Automata-based approach for kernel trace analysis. In *2009 Canadian Conference on Electrical and Computer Engineering*, pages 970–973, May 2009.

[87] J. B. Matthews. What's wrong with c++ templates? "http://people.cs.uchicago.edu/~jacobm/pubs/templates.html", 2017. [Online; accessed 05-May-2017].

[88] I. F. D. Melo, A. Benbachir, and M. Dagenais. Statistical tool to detect software regressions, 2017. [To be published].

[89] Microsoft. Etw. http://msdn.microsoft.com/en-us/library/windows/desktop/aa363668(v=vs.85).aspx., 2017. [Online; accessed 05-May-2017].

[90] E. K. J. L. D. P. A. F. E. B. Mike Y. Chen, Anthony Accardi. Path-based failure and evolution management. In *Proceeding OSDI'04 Proceedings of the 6th conference on Symposium on Operating Systems Design Implementation - Volume 1*, pages 23–23, 2004.

[91] A. Milenkovic. Performance measurements: Perf tool. "http://lacasa.uah.edu/portal/Upload/tutorials/perf.tool/PerfTool.pdf", Jul 2015.

[92] K. Mohror and K. L. Karavanic. Evaluating similarity-based trace reduction techniques for scalable performance analysis. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–12, Nov 2009.

[93] N. Mostafa and C. Krintz. Tracking performance across software revisions. In *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, PPPJ '09, pages 162–171, New York, NY, USA, 2009. ACM.

[94] G. J. Myers. *The art of software testing. 2nd Ed.* Wiley Sons, 2012.

[95] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100, June 2007.

[96] T. H. Nguyen, B. Adams, Z. M. Jiang, A. E. Hassan, M. Nasser, and P. Flora. Automated detection of performance regressions using statistical process control techniques.

In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*, ICPE '12, pages 299–310, New York, NY, USA, 2012. ACM.

[97] T. H. Nguyen, B. Adams, Z. M. Jiang, A. E. Hassan, M. Nasser, and P. Flora. Automated detection of performance regressions using statistical process control techniques. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*, pages 299–310. ACM, 2012.

[98] T. H. D. Nguyen, M. Nagappan, A. E. Hassan, M. Nasser, and P. Flora. An industrial case study of automatically identifying performance regression-causes. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 232–241, New York, NY, USA, 2014. ACM.

[99] U. of Maryland and U. of Wisconsin. Dyninst. `http://www.dyninst.org/`, 2015. Accessed: 2017-04-30.

[100] U. of Waikato. Weka. `http://www.cs.waikato.ac.nz/ml/weka/`, 2017. Accessed: 2017-04-30.

[101] J. Olsa. perf: Add backtrace post dwarf unwind., May 2012.

[102] OpenCV. Difference. `https://github.com/opencv/opencv/compare/3.0.0...3.1.0`, 2012. [Online; accessed 05-May-2017].

[103] OpenCV. Issues 1423. `http://code.opencv.org/issues/1423`, 2012.

[104] Owasp. Source code analysis tools. `https://www.owasp.org/index.php/Source_Code_Analysis_Tools`, 2017. Accessed: 2017-09-30.

[105] V. K. Palepu and J. A. Jones. Visualizing constituent behaviors within executions. In *2013 First IEEE Working Conference on Software Visualization (VISSOFT)*, pages 1–4, Sept 2013.

[106] E. T. G. R. G. Raja R. Sambasivan, Alice X. Zheng. Categorizing and differencing system behaviours. In *SIGPLAN '82 Proceedings of the 1982 SIGPLAN symposium on Compiler construction*, 2007.

[107] A. Rajee and F. S. Francis. A study on outlier distance and sse with multidimensional datasets in k-means clustering, 2013. Fifth International Conference on Advanced Computing (ICoAC).

[108] P. Reynolds, C. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat. Pip: Detecting the unexpected in distributed systems. symposium on networked systems design and implementation. In *Symposium on Networked Systems Design and Implementation (San Jose, CA, 08–10 May 2006)*, pages 115–128, 2006.

[109] D. Robinson. K-means clustering is not a free lunch. `http://varianceexplained.org/r/kmeans-free-lunch/`, 2008. [Online; accessed 05-May-2017].

[110] B. Ryder. Constructing the call graph of a program.software engineering, ieee transactions on, vol. se-5, no.3pp. 216– 226, may 1979, May 2017.

[111] S. S, Shende, and A. D. Malony. The tau parallel performance system. *Journal of Molecular Biology*, 20:287–311, 2006.

[112] R. R. Sambasivan, A. X. Zheng, M. De Rosa, E. Krevat, S. Whitman, M. Stroucken, W. Wang, L. Xu, and G. R. Ganger. Diagnosing performance changes by comparing request flows. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 43–56, Berkeley, CA, USA, 2011. USENIX Association.

[113] K. Scott, N. Kumar, S. Velusamy, B. Childers, J. W. Davidson, and M. L. Soffa. Retargetable and reconfigurable software dynamic translation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '03, pages 36–47, Washington, DC, USA, 2003. IEEE Computer Society.

[114] W. Shang, A. E. Hassan, M. Nasser, and P. Flora. Automated detection of performance regressions using regression models on clustered performance counters. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, ICPE '15, pages 15–26, New York, NY, USA, 2015. ACM.

[115] B. Sharma, P. Jayachandran, A. Verma, and C. R. Das. Cloudpd: Problem determination and diagnosis in shared dynamic clouds. In *Proceedings of the 2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, DSN '13, pages 1–12, Washington, DC, USA, 2013. IEEE Computer Society.

[116] S. Sharma. Fast tracing with gdb. `https://suchakra.wordpress.com/2016/06/29/fast-tracing-with-gdb/`, 2016. [Accessed 05-May-2017].

[117] A. Silberschatz, G. Gagne, and P. B. Galvin. Operating systems concepts essentials, 2006. 1ed. Wiley.

[118] Sourceforge. Findbugs. http://findbugs.sourceforge.net/, 2017. Accessed: 2017-04-30.

[119] Sourceforge. Oprofile. http://oprofile.sourceforge.net/doc/introduction.html, 2017. Accessed: 2017-04-30.

[120] Sourceforge. Pmd. https://pmd.github.io/, 2017. Accessed: 2017-09-30.

[121] Sourceware. Debugging with gdb. https://sourceware.org/gdb/current/onlinedocs/gdb/index.html#SEC_Contents, 2012.

[122] A. Spear, M. Levy, and M. Desnoyers. Using tracing to solve the multicore system debug problem. *Computer*, 45(12):60–64, Dec 2012.

[123] M. Sprunck. Findbugs - static code analysis of java. http://www.methodsandtools.com/tools/findbugs.php, 2012. Methods and tools Magazine.

[124] W. N. Sumner, Y. Zheng, D. Weeratunge, and X. Zhang. Precise calling context encoding, May 2017.

[125] H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. http://www.gotw.ca/publications/concurrency-ddj.htm, 2005. [Online; accessed 05-May-2017].

[126] Sybase. End to end tracing. http://infocenter.sybase.com/help/index.jsp?topic=/com.sybase.infocenter.dc01217.0224/doc/html/nkr1338287526815.html, 2013. [Online; accessed 05-May-2017].

[127] R. Tibshirani and T. Hastie. Virtual machines cpu monitoring with kernel tracing. http://statweb.stanford.edu/~tibs/stat315a/LECTURES/em.pdf, 2008. [Online; accessed 05-May-2017].

[128] R. Tibshirani, G. Walther, and T. Hastie. Estimating the number of clusters in a dataset via the gap statistic. 63:411–423, 2000.

[129] R. A. Vitillo. Performance tools developments.future computing in particle physics, 16 june 2011. http://indico.cern.ch/event/141309/contributions/1369454/attachments/126021/178987/RobertoVitillo_FutureTech_EDI.pdf, May 2011. [Online; accessed 05-May-2017].

[130] VMware. Dynamorio. http://www.dynamorio.org/, 2017. Accessed: 2017-04-30.

[131] K. Waclena. Pattern matching. `http://www2.lib.uchicago.edu/keith/ocaml-class/pattern-matching.html`, 2006. Accessed: 2017-04-30.

[132] M. Weber, R. Brendel, and H. Brunst. Trace file comparison with a hierarchical sequence alignment algorithm. In *Proceedings of the 2012 IEEE 10th International Symposium on Parallel and Distributed Processing with Applications*, ISPA '12, pages 247–254, Washington, DC, USA, 2012. IEEE Computer Society.

[133] I. H. Written and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations. 2nd Edition.* Morgan Kaufmann, 2003.

[134] M. i. S. X. Zhuang, S. Kim and J.-D. Choi. Perfdiff: A framework for performance difference analysis in a virtual machine environment in proceedings of the 6th annual ieee/acm international symposium on code generation and optimization. new york, ny: Acm, 2008, pp. 4–13., Jul 2015.

[135] L. Yang, C. Liu, J. M. Schopf, and I. Foster. Anomaly detection and diagnosis in grid environments. In *SC07*, pages 10–16, Nov 2007.

[136] I. Yanok and N. Nystrom. Tapir: A language for verified os kernel probes. In *Proceedings of the 8th Workshop on Programming Languages and Operating Systems*, PLOS '15, pages 33–38, New York, NY, USA, 2015. ACM.

[137] Z. Zhang and H. Shen. Online training of svms for real-time intrusion detection. In *18th International Conference on Advanced Information Networking and Applications, 2004. AINA 2004.*, volume 1, pages 568–573 Vol.1, 2004.

[138] A. X. Zheng, J. Lloyd, and E. Brewer. Failure diagnosis using decision trees. In *Proceedings of the First International Conference on Autonomic Computing*, ICAC '04, pages 36–43, Washington, DC, USA, 2004. IEEE Computer Society.

[139] X. Zhuang, S. Kim, M. i. Serrano, and J.-D. Choi. Perfdiff: A framework for performance difference analysis in a virtual machine environment. In *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '08, pages 4–13, New York, NY, USA, 2008. ACM.

[140] X. Zhuang, M. J. Serrano, H. W. Cain, and J.-D. Choi. Accurate, efficient, and adaptive calling context profiling. *SIGPLAN Not.*, 41(6):263–271, June 2006.

[141] C. Zou, W. Jiang, and F. Tsung. A lasso-based diagnostic framework for multivariate statistical process control. *Technometrics*, 53(3):297–309, 2011.

APPENDIX

The following work was done in collaboration:
Title: Automated Performance Deviation Detection Across Software Versions Releases
Authors: Abderrahmane Benbachir, Isnaldo Francisco de Melo, Michel Dagenais and Bram Adams. This paper was accepted in the conference Software Quality, Reliability and Security, QRS 2017.
My contribution: data analysis, statistical verification, comparative methods.

Title: Evaluating C++ Performance Code Practices
Authors: Isnaldo Francisco de Melo and Fabio Petrilo. This paper was submitted to the conference , 36th IEEE – International Performance Computing and Communications Conference, IPCCC 2017.
My contribution: performance analysis and evaluation, C++ benchmarks, collection of optimizations.